To the Graduate Council:

I am submitting herewith a thesis written by H. Venkateswaran entitled "Tree-Structured Parallel Algorithms." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

David L. Matuszek

David L. Matuszek, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

L. Evans Peck

Vice Chancellor
Graduate Studies and Research

TREE-STRUCTURED PARALLEL ALGORITHMS

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

H. Venkateswaran

March 1982

3058024

To

Chinnamanni, Appa

and Usha

ACKNOWLEDGEMENTS

Acknowledgements are due to my thesis advisor Dr. David L. Matuszek for the discussions I had with him on this topic. Acknowledgements are also due to Dr. Terry Feagin and Dr. David Straight for their valuable comments. I am grateful to my friend and colleague Mr. Richard L. Christiansen for many helpful suggestions.

I am greatly indebted to Prof. E. V. Krishnamurthy for introducing me to this exciting area of Computer Science.

I am also grateful to Mr. Ashwin Honkan and other colleagues at U.T. Thanks are also due to the Computer Science faculty and office staff who provided a very friendly environment.

Finally I would like to record here my gratitude to Mr. S. Panchapakesan and other colleagues at the National Aeronautical Laboratory, Bangalore, India.

ABSTRACT

This paper presents a study of tree-structured parallel algorithms. Concepts such as flow, synchronous and asynchronous tree algorithms, and speed-up for this class of algorithms are defined. It is shown that under certain reasonable assumptions the speed-up, in going from a sequential to a parallel execution, is given by the ratio of the number of nodes in the tree to the number of levels in the tree. These concepts are then used in the study of algorithms for the following four problems: finding a maximal common subsequence of two given strings, locating the zeros of a complex polynomial in a given rectangle in the complex plane, allocating resources to some n activities maximizing the returns using a dynamic programming approach, and evaluating a multivariate polynomial. The polynomial zeros algorithm is shown to be an asynchronous tree algorithm whose speed-up is given by O(n). The other algorithms are shown to be synchronous tree algorithms and their speed-up is $O(n/\log_2 n)$. These serve as examples of algorithms with an inherent limitation preventing them from achieving a speed-up factor of O(n).

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The design and analysis of parallel algorithms have interested researchers since the early days of computing. The inherent parallelism in algorithms involving matrices, fast Fourier transforms, etc. was identified and studied even before the advent of parallel computers. The development of efficient parallel algorithms has become even more important for the following reasons:

(a) Advances in computer technology have made it feasible to join several processors to function as a single system at low cost.

(b) Even though there is a proliferation of computer systems, the demand for computer time has grown. This is due to several factors such as awareness, increase in the number of application areas and advances in technology. Parallel computing can be very useful in meeting this demand for computers.

(c) The study of complexity of computer algorithms has provided a basis for classification of algorithms according to suitably chosen complexity measures. This has led to the study of efficient parallel realizations of many of these algorithms and the effect of parallelism on their complexity.

We list below some of the questions being addressed in the study of parallel algorithms. These cover a wide range of issues including theoretical and practical aspects of this area.

(1) Do they fall into finitely many categories with respect to their structure? That is, does there exist a taxonomy of parallel algorithms? Are there optimal structures for parallel algorithms?

(2) What are the components in the complexity measure for parallel algorithms? What are the trade-offs involved in going to a parallel algorithm from a sequential algorithm for the same problem in terms of efficiency gain and resources required?

(3) How can the correctness of parallel algorithms be established? What model is convenient for this aspect? What abstract model captures the essence of parallel computation?

(4) How can parallel algorithms be implemented? How are algorithms matched with architectures? How are parallel tasks assigned to processors?

(5) Are these expressible naturally using the currently available language structures?

(6) Can parallelism be viewed as a programming technique such as recursion? What is the relationship between parallelism and programming paradigms such as Divide and Conquer, Balancing, Nondeterminacy and Dynamic programming?

See Stone [48] for an early discussion on the problems of parallel computation.

It is our objective here to examine some of the well-known parallel algorithms with respect to their structure. We concern ourselves with algorithms which have an inherent tree structure. Trees are basic computational structures arising in many applications. In the algorithmic solution of many problems they occur naturally either in the actions constituting the algorithm or in the data on which the algorithm operates, or in both. They are also used to model computations performed or data organized in a hierarchical fashion.

Clearly there are many parallel algorithms with structures other than a tree. The flow graph of the algorithm for computing the length matrix in the common subsequence problem is not a tree (as we show in chapter 4). The garbage collection algorithm of Dijkstra et al. [15], concurrent operations on B-trees described by Bayer and Schkolnick [6], concurrent operations on AVL trees described by Ellis [16] and concurrent operations on data bases described by Bernstein and Goodman [9] are other examples of parallel algorithms which do not exhibit a tree structure. But there is a large class of problems for which application of programming techniques like partitioning (of the action space or data space) lead naturally to tree structured algorithms.

Also, the realization of tree structured parallel algorithms has received considerable attention. Many architectures especially suitable for implementing such algorithms have been proposed. These machines have been called "tree machines". See, for instance, Benteley and Kung [7], for a discussion on a binary tree machine on a single chip suitable for search type operations on trees. Machines proposed by Despain and Patterson [14], and Harris and Smith [20] are other examples. See also Horowitz and Zorat [23] for a discussion on binary tree machines.

The organization of the thesis is as follows: In Chapter 2, we present a general discussion on structure and complexity concerns involved in parallel computations; in Chapter 3, we discuss tree structured algorithms and define the concepts used in the study of specific parallel algorithms. In subsequent chapters, these concepts are applied to particular algorithms for problems such as maximal common subsequence of two strings, polynomial zeros, resource allocation using dynamic programming, and multivariate polynomial evaluation. The thesis is concluded with an appendix containing a bibliography of several published parallel algorithms.

CHAPTER 2

STRUCTURE AND COMPLEXITY OF PARALLEL ALGORITHMS

2.1 STRUCTURE

A study of the structure of parallel algorithms is useful for:

(a) algorithm design - for identifying parallelism in existing algorithms and the design of new parallel algorithms.

(b) efficient implementation of algorithms on existing parallel computer systems.

(c) motivating the design of novel computer architectures.

(d) studying properties of parallel algorithms - including their complexity and correctness.

(e) design of language structures supporting parallel computation.

Several parallel algorithms have been described in literature (See Appendix for a list containing many published parallel algorithms). But there have not been many attempts at providing a taxonomy. As part of the work on identifying parallelism in 'ordinary' programs, Kuck [33] has studied the structure of programs and how they relate to parallelism. There are many parallel algorithms which can

be cast in the form of evaluation of recurrence equations. These have been studied extensively; see, for instance, Kuck [33] and Kogge [30]. A class of methods known as recursive doubling has been defined for such problems by Stone [48]. Recursive doubling is defined as that process in which at every step a computation breaks up into two independent computations. The underlying structure of these algorithms is a binary tree.

In a recent survey on the structure of parallel algorithms, Kung [37] identifies three independent aspects of a parallel algorithm, namely, module granularity, concurrency control and communication geometry. Module granularity is defined as the amount of computation that can be performed in each independent subtask of a task. Concurrency control refers to the synchronization required between the independent subtasks. Communication geometry is the geometry resulting from the layout of independent subtasks and the flow of information between them. These are then used to provide a taxonomy of parallel algorithms. Kung [37] also provides a table of the known parallel algorithms classified using this as a basis.

Essentially, all these classifications relate each parallel algorithm to parallel architectures to which it corresponds naturally. A classification of computer systems by Flynn [18] serves as a basis for this.

Parallelism in computer systems arise from their capability to process concurrently,

1. single instruction stream on multiple data stream (SIMD).
2. multiple instruction stream on single data stream (MISD).
3. multiple instruction stream on multiple data stream (MIMD).

The realization of this parallelism may be achieved by using different architectures. With the advent of LSI chips, it has become possible to build systems consisting of several processors inexpensively.

Parallel algorithms have been classified into two categories depending upon the amount of synchronization required between the independent subtasks of a task: Synchronous and Asynchronous algorithms.

2.1.1 Synchronous Algorithms

Under this category come all those algorithms in which there is extensive communication between the independent subtasks. In Kung's [37] terminology, these are algorithms with high concurrency control and low module granularity. Corresponding naturally to parallel architectures there are three types of synchronous algorithms: SIMD, MISD and systolic algorithms.

In SIMD (Single Instruction Multiple Data) algorithms, identical operations are performed on different sets of data concurrently. Most of the currently known algorithms are of this type. Examples: algorithms for sorting such as

Mergesort, Quicksort and Heapsort; polynomial evaluation, polynomial zeros, matrix operations, FFT and binary tree traversal.

In MISD (Multiple Instruction Single Data) algorithms, different operations are performed on the same set of data. Examples: algorithms used by a scheduler in a computer system which uses a priority queue data structure; use of a matrix 'A' in different operations such as decomposition and multiplication.

In systolic algorithms, each independent task pumps data in and out regularly (in constant time intervals), each task performing some short computation. Examples: Inner product of two vectors, LU decomposition of a matrix.

2.1.2 Asynchronous Algorithms

All those algorithms in which very little synchronization is required between independent subtasks are referred to as asynchronous algorithms. There are two types of asynchronous algorithms corresponding to two types of parallel architectures: MIMD and data-flow.

In MIMD (Multiple Instruction Multiple Data) algorithms, different operations are performed on different sets of data concurrently. The design of these algorithms is one of the challenging problems in the study of parallel algorithms. Examples: Garbage collection, concurrent operations on a binary tree or a B-tree, concurrent operations on data bases and iterative schemes in numerical analysis.

In data flow algorithms, each computation is represented as a node with input and output edges. A computation is executed (or fired) as soon as all the inputs are ready. The concept of a variable is absent in this type of computation. It is said to be functional in nature. Investigations are under way for practical realization of data flow architectures and languages. See, for instance, Dennis and Misunas [13], and Ackermann [1].

Some of the algorithms in one class can be viewed as belonging to another class depending upon how one interprets the terms 'instruction stream' and 'data stream'. For instance, concurrent operations on a binary tree such as in-order traversal and balancing can be considered as an MISD algorithm if the data-structure binary tree is viewed as a single data entity. It can be considered as an MIMD algorithm if we view the binary tree as two different entities for the two operations. A similar example is the searching of a branch of a binary tree for an element while inserting an element in the other. Consider the case of an SIMD algorithm such as the polynomial zeros algorithm (see chap. 5). Once two subrectangles of a given rectangle are identified, there is absolutely no interaction between the processors operating on the subrectangles though they execute the same instructions. Having copies of the instruction stream in the processors would render this algorithm into an asynchronous algorithm of the MIMD type.

Measures for module granularity and concurrency control would therefore be useful in fixing the classes.

A classification of parallel algorithms relating them to parallel architectures can be shown as in Figure 2.1.

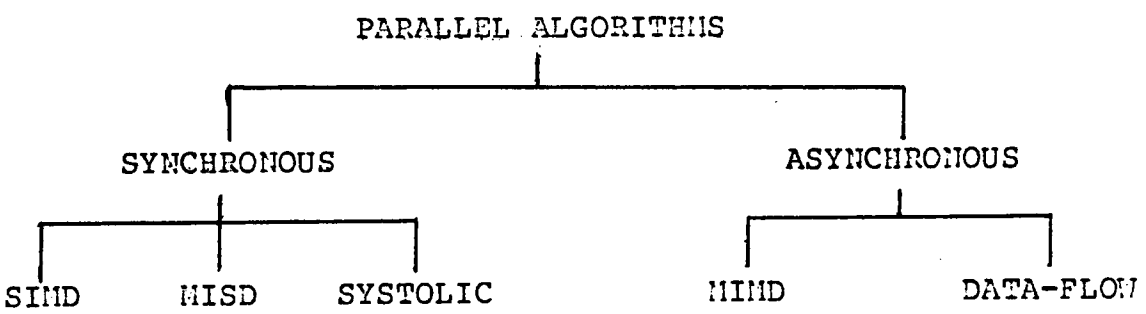

Figure 2.1 Classification of Parallel Algorithms.

Other classifications of parallel algorithms are possible by analyzing their inherent structure. Here, we have attempted a study in this direction by looking at some aspects of algorithms which exhibit a tree structure.

2.2 COMPLEXITY CONSIDERATIONS IN PARALLEL COMPUTATIONS

The complexity of an algorithm estimates the number of operations and the amount of space that would be required for its execution. This demand on resources made by an algorithm is an inherent property of the algorithm independent of the particular machine on which it is implemented (of course, we refer to machines which can be built in practice). But analysis of algorithms for such

properties requires a frame of reference with respect to which one can measure the time and space taken by an algorithm. This is provided by an abstract computational model with characteristics of practical computer systems. Depending upon the nature of study this model may be different from one analysis to another. A good introduction to models of computation useful for analyzing algorithms is presented in Aho et al. [2]. The complexity of an algorithm is measured in terms of the number of operations that it would take on the chosen computational model and the amount of space that is required on it.

An algorithm being a schema of computation, its analysis depends upon the interpretation given to its actions and data. The number of operations may vary from one interpretation to another. It may not be possible to get an exact count of the operations involved in executing an algorithm. But in most cases it is possible to get an estimate for this as a function of certain variables in the problem which the algorithm solves. For instance, in the problem of sorting a set of numbers, the number of elements in the set is a variable in terms of which we can express the number of operations. This also requires that we assume that the operation we are interested in is a comparison and that it takes unit time on the computational model to compare two numbers of the size present in the set. Similarly, in most graph algorithms the complexity can be expressed in terms of the number of vertices and the number

of edges in the graph. The size of a problem is represented by an integer as a function of which the complexity of an algorithm for solving the problem can be expressed. In many instances, it may be required to get only an order bound on the number of operations as this will give estimates on the demands of the algorithm as the problem size increases. The order notation for expressing these bounds has been discussed by Knuth [29].

A function f(n) is said to be bounded above by g(n), i.e., $f(n)=O(g(n))$ if and only if there exist two positive constants c and n_0 such that $|f(n)| < c\,|g(n)|$ for all $n \geq n_0$. This expresses an upper bound. f(n) is said to be bounded below by g(n), i.e., $f(n) = \Omega(g(n))$ if and only if there exist two positive constants c and n_0 such that $|f(n)| > c|g(n)|$ for all $n \geq n_0$. In cases where there exist positive constants c_1, c_2 and n_0 such that for all $n > n_0$, $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$, $f(n) = \Theta(g(n))$.

In the case of parallel algorithms we use the notation $O(g(n),p)$ to denote an upper bound. Here p is the number of processors used.

Sequential algorithms have been extensively studied with respect to their time and space requirements under suitable models of computation. See, for instance, Aho [2] and Weide [51]. Every algorithm can be viewed as a point in the two dimensional space of Time x Space. For several algorithms it is possible to trade time for space or space for time depending upon the constraints on these two

resources. This constitutes a movement in the two dimensional space in an attempt to locate a position satisfying the constraints. These time and space trade-offs have been analyzed under idealized assumptions by several researchers; see, for instance, Savage [44]. Such a study for FFT algorithm is presented by Savage and Swamy [45]. Dynamic programming (DP) is a classic example of a trade-off where space is traded to gain in time efficiency. Consider the problem of optimum associativity in forming a sequence of matrix products. The approach of finding out optimal pairings by examining all combinations yields an exponential algorithm, whereas a DP approach in which the optimal pairing is found out in stages using information already obtained gives an $O(n^3)$ algorithm, see, for instance, Aho et al. [2]. Such modifications are not always possible. For instance, consider the class of NP-hard problems. Algorithms for problems in this class requiring exponential time with a deterministic computational model can be transformed only, if at all possible, to ones requiring exponential space.

It is usual to consider time as a more critical resource than space and much effort is devoted to trading space for time because space requirements are usually linear and almost never worse than quadratic.

In the study of parallel algorithms there is one more dimension to be considered corresponding to the number of processors used. Every parallel algorithm can be viewed as

a point in the three-dimensional space of Time x Space x Processors. The scope for trade-offs is much greater. It is not always the case that using n processors to work on an algorithm would reduce its complexity by a factor of n. For many problems a lower bound analysis with respect to parallel computation has shown that it is due to an inherent limitation of the problem. For many other problems this analysis does show that the complexity could be reduced by a factor of n and it is a matter of devising an algorithm. This could be very difficult, as has been the case with sequential algorithms where attempts are being made to achieve the efficiency limits indicated by lower bounds for several problems. For discussions on lower bounds as applicable to parallel computations, see Borodin and Munro [11], and Horowitz and Sahni [22].

For examples of problems with inherent limitation see Valiant [49] who discusses problems involving binary comparisons, Hyafil and Kung [25] who discuss solutions of linear recurrence equations and Kung [35] who discusses parallel evaluation of rational expressions. See also Kung [36]. We also refer the reader to references given in these papers.

One of the major factors limiting the efficiency of a parallel algorithm from reaching theoretical lower bounds is the time taken for communication among the parallel parts of an algorithm. Lint and Agarwala [39] discuss this problem in detail and present examples to show how a rearrangement

of computation and/or data might help in achieving the lower bounds in some cases. The communication time between the parallel parts of an algorithm is dependent on the amount of information to be transmitted and the arrangement of the parallel processing elements. Several interconnection structures have been studied to minimize this communication delay.

The complexity of parallel computations has been studied under different models of computation. There are three aspects of a model for this purpose.

(1) Number of processing elements. There are two types of models with respect to the number of processors available as a parallel computer system: the unbounded parallel model and the K-processor parallel model. The unbounded model assumes the availability of an unlimited number of processors. Usually, it is a function of the size of the problem under analysis. For instance, for the Mergesort technique the unbounded parallel model may assume the existence of a maximum of O(n/2) processors where n is the number of elements to be sorted. The K-processor model assumes the availability of only a fixed number K of processors independent of the size of the problem being solved on it.

(2) Functional capability of the processors. This concerns the assumptions about the capability of the individual processors used as a parallel computer system. This concerns issues such as whether the processors are identical

functionally, whether they perform short computations requiring constant time, etc. Several types of models have been studied with respect to the functions of the processors constituting the system. These depend upon the problems and the methods used to solve them.

(3) Communication structure between the processors. This addresses the issues concerning communication among the different processors in a parallel computer system. The considerations include the amount of information transfer between different processors during the course of an algorithm, the delay for this transfer, and the complexity of routing information from a source processor to a destination processor. Several interconnection structures have been studied with respect to these characteristics, among others such as connection costs and redundancy in connection pattern. See, for instance, Anderson and Jenson [3], and Wittie [53].

CHAPTER 3

TREE STRUCTURED ALGORITHMS

In this chapter we give some definitions and results useful in discussing tree-structured algorithms. We then consider data organizations and programming techniques which generate tree structured algorithms.

3.1 DEFINITIONS AND TERMINOLOGY

Tree. A tree is a finite set of nodes such that (i) there is one specially designated node called the root; (ii) the remaining nodes are partitioned into n>0 disjoint sets T1,T2,...,Tn, where each of these sets form a tree, called the subtrees of the root.

Degree. The number of subtrees of a node is called its degree. The degree of a tree is the maximum degree of the nodes in the tree. Nodes that have degree 0 are called terminal nodes or leaves. The roots of subtrees of a node are called its children and a node is called the parent of the roots of its subtrees.

Level. The root of a tree is said to be at level 1. If a node is at level p, then its children are at level p+1.

Height. The maximum level of any node in the tree is called the height (or depth) of the tree.

Binary Tree. A finite set of nodes which is either empty or consists of a root and one or two disjoint binary trees is called a binary tree.

Complete Binary Tree. A binary tree of height m which has exactly $(2^{m+1}-1)$ nodes is called a complete binary tree. The number of leaves in such a tree is (2^m).

Balance. The balance of a node in the binary tree is defined to be the difference between the height of its two children. The balance of the binary tree is the balance of its root.

Action. By the term action, we mean the program (algorithm) segment that is executed by a single processor. This may range from elementary operations to high level control structures, depending upon the level of abstraction. For instance, in the resource allocation problem (ch. 6) an action is maximizing a set of functions; in the mergesort problem (Section 3.3.1) an action is sorting a set of numbers. The meaning associated with the word action will be clear from the context. Also we assume that an action manipulates some data, i.e., all actions need some data.

Parallelism in a Tree. Two children of the same node are said to be parallel to each other.

Degree of Parallelism in a Tree. This is defined to be the number of leaves in the tree. For a complete binary tree with m levels the degree of parallelism is 2^m. For a tree structured algorithm, this is the maximum number of parallel actions which may be generated in the algorithm.

If a single processor is associated with each of these parallel actions the degree of parallelism is the maximum number of processors that may be used in parallel in the execution of the algorithm. An algorithm is sequential if its degree of parallelism is 1 and is parallel if the degree of parallelism is > 1.

Every algorithm has associated with it two types of structures corresponding to its control flow and data flow. The control flow structure determines the relation of actions in the algorithm ordered with respect to time. The data flow structure determines the dependency of data objects manipulated by the algorithm.

Example. Consider the evaluation of the product of n matrices M = M1 x M2 x...x Mn where each Mi is a matrix with r rows and r columns.

A dynamic programming algorithm for determining the optimum ordering for multiplying the matrices is given as follows by Aho et al. [2]:

```
BEGIN
        (1) FOR  i :=  1 UNTIL n DO
                    m_ii := 0;
        (2) FOR p :=  1 UNTIL n-1 DO
                   FOR i := 1  UNTIL n - p  DO
                     BEGIN
                      j :=   i + p;
                      m_ij :=  Min  (m_ik + m_k+1,j + r_i-1 * r_k * r_j )
                               i≦k≦j
                     END
END
```

m_{1n} gives the minimum number of operations to get M. Let n=4 so that M=M1 x M2 x M3 x M4. Let us treat the computation of m_{ij} for a fixed i and j as an action and consider the control flow structure and data flow structure of the algorithm. Let the computation of m_{ij} be represented by a node labelled m_{ij} in the following figures. The control flow structure is shown in Figure 3.1.

Figure 3.1 Control Flow Structure.

Now, $m_{12} = r_0 r_1 r_2$, $m_{23} = r_1 r_2 r_3$, $m_{34} = r_2 r_3 r_4$, m_{13} = Min $(m_{12} + r_0 r_2 r_3, m_{23} + r_0 r_1 r_3)$, m_{24} = Min $(m_{23} + r_1 r_3 r_4, m_{34} + r_1 r_2 r_4)$, m_{14} = Min $(m_{13} + r_0 r_3 r_4, m_{12} + m_{34} + r_0 r_2 r_4, m_{24} + r_0 r_1 r_4)$.

Here the data for the actions m_{12}, m_{23} and m_{34} are not generated by any of the other actions (but available as data to the problem). m_{13} is data-dependent on m_{12} and m_{23}, m_{24} is data-dependent on m_{23} and m_{34}, and m_{14} is data-dependent on

m_{12},m_{13},m_{24} and m_{34}. So, the data flow structure with respect to these actions is as in Figure 3.2.

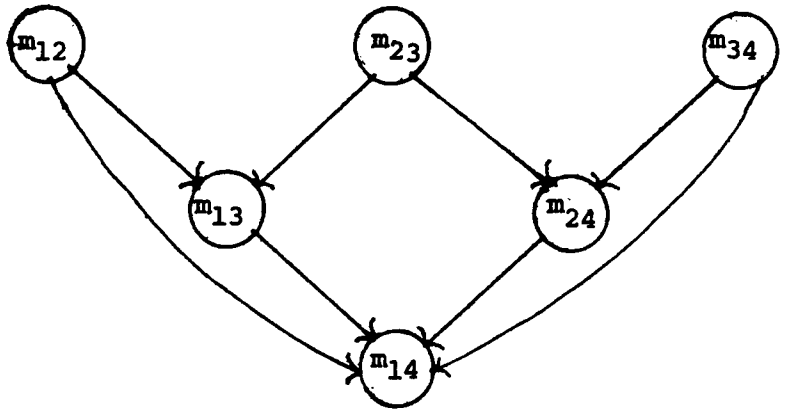

Figure 3.2 Data Flow Structure.

Control Dependency. An action P is control-dependent upon another action Q if P can be performed only after Q has completed. An action P can be control dependent on more than one action Q1,Q2,...,Qn. The control dependency graph of an algorithm in which the nodes are actions and the edges connect actions which are control-dependent, is called its control flow structure. In the special case that each action is control-dependent on only one other action this structure will be a tree, as each node will have at most a single parent. Several actions may be control dependent upon a single action represented by branches from a node to several nodes.

Data Dependency. An action P is data-dependent upon another action Q if the data used by P is generated by Q. An action P can be data dependent on more than one action

Q1,Q2,...,Qn. The data dependency graph of an algorithm in which the nodes are actions and edges connect actions which are data-dependent, is called its data flow structure. This data flow graph has also been called the communication geometry of the algorithm by Kung [37]. In the special case that each action is data-dependent on only one other action this structure will be a tree. Several actions may be data-dependent on a single action.

In the case of sequential algorithms the order in which actions are presented determines the control flow structure and this may be different from its data flow structure. In parallel algorithms, assuming that an unlimited number of processors are available as needed and all actions need some data, an action P can be performed when its data is ready. This implies that if an action P is control-dependent on another action Q then it must wait for its data to be generated by Q, i.e., P is data-dependent on Q. Similarly, if P is data-dependent on Q it is also control dependent on Q. Hence under the assumption of parallelism the control flow structure will be the same as data flow structure, and we will refer to this in the sequel as the structure of the algorithm. Similarly, in what follows we will use the term dependency to mean both control-dependency and data-dependency.

Flow. Let T=(A,E) be a rooted tree where A is a set of nodes representing the actions in the algorithm and E is a set of branches representing the transition between actions.

The set A consists of a distinguished node called the root. Let T be a K-ary tree, i.e., from any node in the tree there are at most K branches. Let there be (p+1) levels in the tree. Then the set A is partitioned into (p+1) subsets A0,A1,...,Ap where Ai consists of the nodes at level i in the tree. Flow in T can be defined as consisting of two components:

(a) A relation between the actions in a set Ai with the actions in another set Aj. This relation is called 'succession' (and the nodes of Aj succeed those in Ai) since for every action Ajk in Aj there is an action in Ai which Ajk strictly follows in time.

(b) Flow direction which decides whether i<j or j<i in this relation. Since T is a tree the flow direction determines a partial ordering of the set A.

If i<j then the flow is said to be 'forward' as it proceeds from the root to the leaves. In this case the flow relates a single action with multiple actions.

If i>j then the flow is said to be 'backward' and it proceeds from the leaves to the root. The relation is many to one.

Fork and Join. In the case of forward flow, a node may have many successors. This is a point of generation or a 'fork' in the flow of the tree. Similarly, in the case of backward flow, a node encountered represents an action which has to wait for the completion of many actions before it can begin. This is a point of synchronization or a 'join' in the flow of the tree.

Asynchronous and Synchronous Tree Algorithms. An algorithm which has a forward flow in its tree structure is called an asynchronous tree algorithm and one which has a backward flow is called a synchronous tree algorithm.

Many recursive algorithms have two phases. In the first phase the actions corresponding to children nodes are generated during a forward flow. When the actions corresponding to the leaves of the tree are generated the second phase of the algorithm begins in which the actions are executed during a backward flow. Such algorithms which are partly asynchronous and partly synchronous are referred to as synchronous tree algorithms.

Efficiency Gain. Let T(A,E) be a tree where A consists of N nodes representing the actions in an algorithm and E is a set of branches representing the transition between actions. Let T have L levels. Let each action take t units of time uniformly. If there is only one processor available the execution of the algorithm would take Nt units of time. If a sufficient number of processors is available the execution would take Lt units of time. This motivates the definition of the efficiency gain factor in going from sequential to parallel execution for tree structured algorithms. The efficiency gain factor is defined as the ratio of the difference of the number of nodes in the tree and the number of levels to the number of nodes. That is, efficiency gain is (N-L)/N. This gives a measure in a scale

of 0 to 1. The ratio of the time for parallel execution to the time for sequential execution has been referred to as speed-up by Kuck [33]. With the stated assumptions the speed-up for tree structured algorithms is N/L.

Module Granularity. The time taken for the execution of an action associated with a node is defined to be its module granularity. This is measured in terms of the number of operations required for performing the action. An operation here is defined as one which takes unit time with respect to a computational model.

Information Weight. The amount of information transferred between two nodes during a flow is called the information weight associated with the branch linking the two nodes. This is measured in terms of the number of bits in the information if a logarthmic cost criterion is used or in terms of the number of units of information if a uniform cost criterion is used. Assuming that it takes unit time to transmit unit information from one node to another the information weight measures the communication time between two actions.

For instance, consider the problem of evaluating a polynomial of degree 7. The actions for evaluation can be represented as a binary tree as in Figure 3.3.

The three levels in the tree partition the set of actions into the subsets A0,A1,A2. The set A0 consists of 1 action A00 which is to perform: first value.x + second

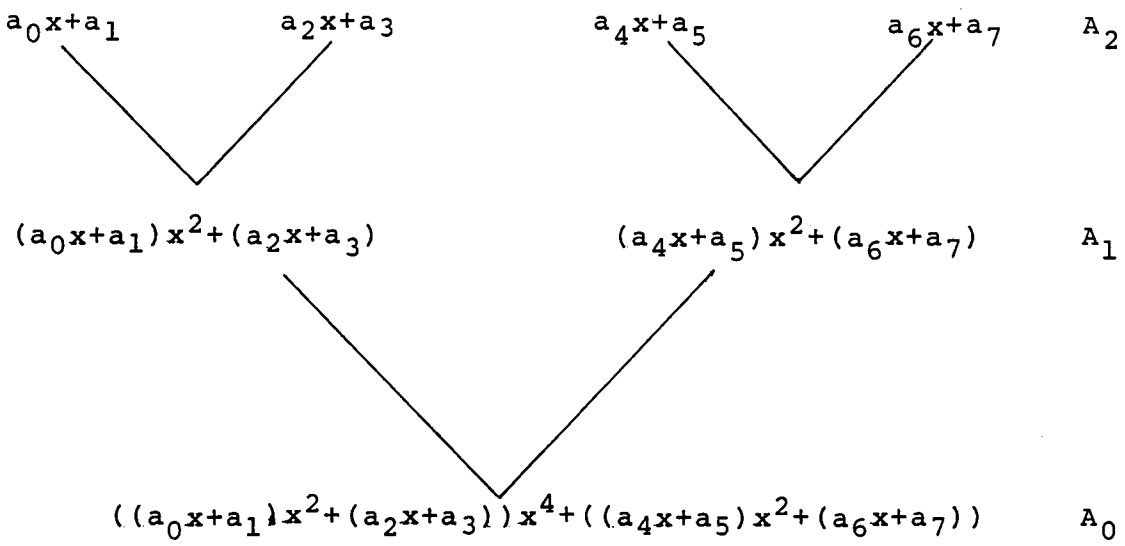

Figure 3.3 Polynomial Evaluation.

value. The set A1 consists of 2 actions A10,A11 where both of them perform: first value.x^2+ second value. The set A2 consists of 4 actions A20,A21,A22,A23 which perform: first value.x + second value.

The flow for this tree is defined by the relations A2 A1 and A1 A0. The flow direction is backward. This is an example of a synchronous tree algorithm. The total number of actions in this tree is 7. The number of levels is 3. Hence, the efficiency gain in going to parallel execution from a sequential execution is 4/7. The speed-up is 7/3. The degree of parallelism is 4, which is the number of leaves in the tree; this is the maximum number of processors which can perform actions in parallel. There is one value transmitted from each of the nodes at one level to the nodes at the next level.

3.2 TREE STRUCTURED DATA

The organization of data as trees, especially binary trees, and operations on them have been extensively studied. Several binary tree structures for data have been proposed with the objective of realizing operations on them efficiently. Bounded binary trees, AVL trees and B-trees are examples of binary tree organization of data with specific characteristics. In these organizations the concept of balancing is usually introduced so that the data objects are evenly distributed in the tree, making it possible to perform certain operations independent of others. Typically, in these binary trees operations such as insertion, deletion, modification, location and balancing take $O(\log_2 n)$ time because of this independence. See Knuth [27] for an in-depth treatment of these topics.

Though it is possible to access several data objects in a tree simultaneously, parallelism is very dependent on the structuring of operations on these objects. It is necessary to ensure that concurrent operations preserve data integrity and also leave the tree with the same characteristics for later operations. The coordination required between the concurrent operations for manipulating B-trees have been studied by Bayer and Schkolnick [6], and Samadi [43]. These problems for AVL trees are discussed by Ellis [16]. It is also possible that the actions in an algorithm operating on tree structured data are inherently sequential, thus do not exploit the parallelism in data.

Consider the problem of sorting n numbers. It is well known that any algorithm for this using comparison operations will take at least $O(n \log_2 n)$ time. Heapsort is a technique which achieves this lower bound. It operates by organizing the n numbers as a binary tree satisfying a property called the heap property. A binary tree is said to have the heap property if all its nodes have the heap property. A node has the heap property if either it is a leaf or the value associated with it is not less than the values associated with its children. The heapsort algorithm first represents the given set of numbers as a binary tree and makes it into a heap starting from the parent nodes of the leaves and traversing up the tree up to its root. See Aho et al. [2] for details of the algorithm. This process takes $O(n \log_2 n)$ operations. The root node of the heap tree contains the largest element. It is removed from the root and a leaf put as the root. The tree is again made into a heap. This does not involve all the nodes but only the nodes in the path from the root to a leaf. This takes $O(\log_2 n)$ operations. Thus in every pass one element of the largest magnitude is removed from the tree, and there are n such passes, taking a total of $O(n \log_2 n)$ for sorting.

The construction of the heap initially involves operations on all the elements at a level independent of each other so that it can be done in parallel taking $O(\log_2 n)$ operations with at most $O(n/2)$ processors. But making a heap after removing an element from the heap tree

is sequential because it looks for a single element. Thus it takes $O(n \log_2 n)$ operations also.

3.3 PROGRAMMING PARADIGMS AND TREE STRUCTURE

There have been attempts to identify the fundamental concepts involved in the design of computer algorithms. This has been useful in the study of properties of algorithms. Many of these design methods generate algorithms with a tree structure. For discussions on these methodologies and examples, see Aho et al. [2], Horowitz and Sahni [22] and Knuth [27,28]. Tree structured algorithms have parallelism built into them as actions represented by the branches can be done in parallel. As such, building a tree structure into an algorithm seems to be a parallelizing technique. We attempt to identify here some of the programming techniques which generate a tree structure. Detailed discussions on the various aspects of these techniques are presented in the references cited.

3.3.1 Divide and Conquer

In this approach, a problem is divided into smaller parts and solutions are sought for these smaller and (probably) simpler subproblems. The solutions are then combined to get the solution of the original problem. The subproblems themselves can in general be further subdivided into smaller parts. Representing the original problem as the root of a tree, the subparts can be represented as

children of the root. A node representing a subproblem can be viewed as the root of a tree whose branches lead into nodes representing its subproblems. The actions associated with the children of a node are probable candidates for parallel execution.

Often the subproblems obtained by dividing a problem are of the same type as the original problem. The solution procedure in these cases turns out to be recursive in nature. A divide and conquer strategy in which at every stage a problem is split into two subproblems of the same type has been referred to as recursive doubling.

The computing time of a problem P divided into k parts P1,P2,...,Pk is the sum of the computing times of the subproblems Pi and the time required to combine the solutions. For instance, in the case of recursive doubling algorithms the computing time can be expressed by the recurrence relation,

$$T(n) = \begin{cases} g(n) & \text{for n small} \\ 2T(n/2)+f(n) & \text{otherwise} \end{cases}$$

where T(n) is the time for the original problem with n inputs, g(n) is the time to compute the actions represented by the leaves and f(n) is the time for combining two subsolutions.

There are several problems for which the divide and conquer techniques have been applied, such as binary search, external sorting, bisection method of locating the zero of a

function, and matrix multiplication using Strassen's method. See the references for more details. We discuss some of the recursive parallel algorithms generated by this approach in later chapters.

We present here two short examples of this method applied to sorting numbers. Let S be a set of N numbers to be sorted into ascending order.

Example 1. Merge Sort

Given two sorted lists, there exists a linear algorithm for merging them into a single list. The problem of sorting the set S can be divided into sorting two partitions of S and merging them together. The algorithm for this is:

```
MERGESORT(M,N);  (*This sorts the N elements in the set S
starting from position M by dividing S into two halves, sorting
these two parts and merging them together*)
 BEGIN
    IF N > 1 THEN
         BEGIN
                 MERGESORT(M, N/2 );
                 MERGESORT(M + N/2 , N/2 );
                 MERGE (M,N);
         END
 END MERGESORT.
```

Here procedure MERGE merges two sets of approximately N/2 elements each starting from position M in the set S. The given set is sorted by using the algorithm for (S,1,N). Mergesort can be represented as a binary tree with a node representing the sorting of N numbers from position M and two branches from it leading to nodes representing the sorting of two halves of the set starting from positions 1 and M+ N/2 . For instance, when N=16, the tree would be as in Figure 3.4.

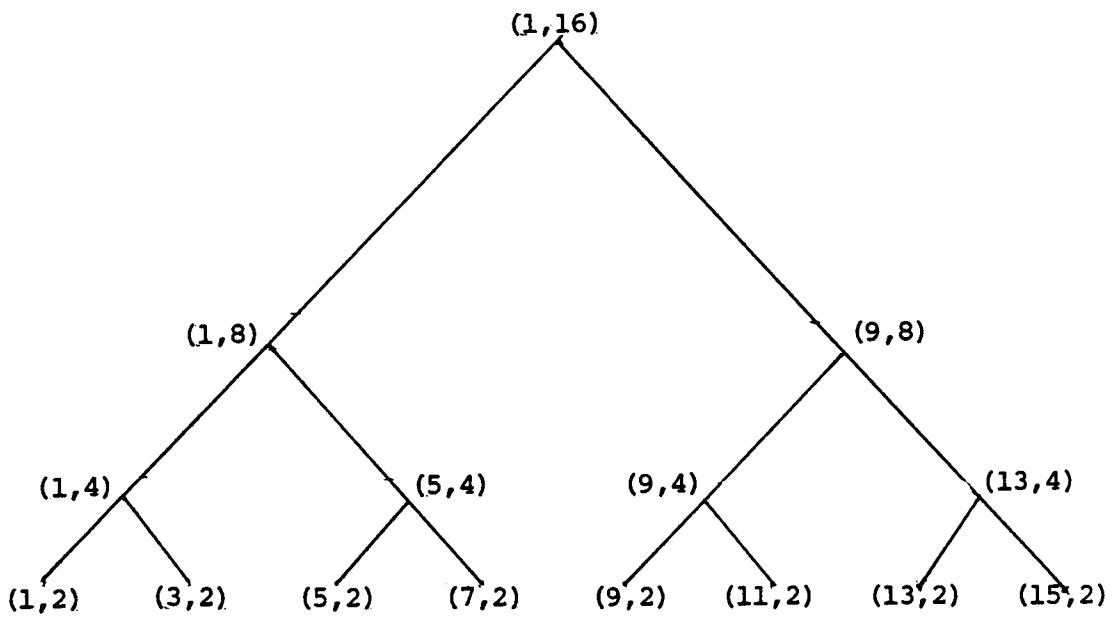

Figure 3.4 Tree for Merge Sort.

This binary tree is first generated by the algorithm, and when the leaves are generated control flows from the leaves to the root by merging the results of two children nodes into a parent node. It is an example of a synchronous parallel algorithm.

Example 2. Quick Sort

In this method, a random element x is chosen from the ordered set S. S is then partitioned into three subsets SL, SE, SH where SL contains all elements of S less than x, SE contains all elements of S equal to x and SH contains all elements of S greater than x. S is now ordered so that elements of SL precede those of SE which precede those of SH. Since the element x (which is in the set SE) has found its position in the ordered set, the original problem reduces to that of sorting the sets SL and SH. The same partitioning technique can be used for this. This generates a binary tree structure. The algorithm can be written as:

```
QUICKSORT(S,N);
 (*This sorts the N elements in the set S by:
      .choosing a random element x in S and partitioning S
       into three sets SL, SE and SH
      .sorting SL and SH *)
BEGIN
        IF  N > 1  THEN
                BEGIN
                    partition S; Let SL contain NL elements
                                 and SH contain NH elements;
                    QUICKSORT(SL,NL);
                    QUICKSORT(SH,NH)
                END
END QUICKSORT.
```

The balance of the binary tree structure in Mergesort is determined by the number of elements in the set S. That is, if there are N=2 elements in the set S then Mergesort generates a complete binary tree. But the balance of the binary tree structure in Quicksort is determined by the relative magnitude of the element chosen at every stage. For instance, let S = (19,23,5,2,4,9,7,6) and let x=9. Then SL = (5,2,4,7,6), SE = (9) and SH = (19,23).

3.3.2 Recursion

Recursion is another powerful programming technique. As Knuth [27] observes, it is an innate characteristic of tree structures. Many tree operations such as searching and traversing are inherently recursive operations.

The use of recursion involves tree structures in two distinct stages. In the first stage a tree is generated by the recursive invocation of a procedure until the basis case is encountered. The initial invocation represents the root and the basis cases represent the leaves of this tree. In the second stage control flows from the leaves to the root of the tree.

There is one essential difference between the tree generated by the divide and conquer approach and recursion. In the former case it is possible that the actions corresponding to the children nodes are different from those corresponding to the parent nodes. In the case of recursion the actions associated with the children nodes are the same

as those associated with the parent nodes. But in practice, the divide and conquer approach generates mostly algorithms which are also recursive in nature. The Quicksort and Mergesort techniques described in the previous section are examples of such classes of algorithms.

Though recursive procedures have an underlying tree structure, not all of them are realizable in parallel. Many of them are strictly sequential in nature. This is because the tree structure generated by them may be such that every node has a single child node. The recursive algorithm for the Tower of Hanoi problem is an example of an algorithm which is strictly sequential in nature.

3.3.3 Dynamic Programming

Many computational problems can be posed as optimization problems. Dynamic Programming is a technique of problem solving based on the principle of optimality which is that an optimal solution obtained in stages has the property that whatever is the state of the solution at an intermediate stage, the solution in the remaining stages must be optimal.

See references cited at the beginning of this chapter for detailed discussions on dynamic programming techniques as applied to several computational problems.

The recursive application of the optimality principle results in a recurrence relation which is solved by a dynamic programming algorithm for the particular problem.

Basically, this approach has two characteristics:

(1) Breaking the solution procedure into stages.

(2) Use of results obtained at a particular stage in subsequent stages.

As a consequence, at any stage of the solution procedure only those which are optimal up to that point are retained. In certain applications of dynamic programming the solution procedure can be divided into independent stages and combined later. This gives rise to tree structured algorithms, as we show in Chapter 6.

3.3.4 Nondeterministic Computations

Problems involving nondeterministic computations also generate a tree which has been variously called a search tree, state space tree, etc. In this tree a node represents a state of computation and a branch represents a transition from one state to another. Nondeterminism is the situation in which there is more than one valid transition from a state of computation. The right transition to take at a particular node may not be evident until after many transitions. One of the approaches taken in such computations is called backtracking. It is depth first traversing of the search tree. This is a recursive procedure which can be written as follows:

```
TREESEARCH(node);
  IF the node is a leaf THEN
        IF the leaf is a goal node THEN report success
        ELSE report failure
  ELSE (* the node has one or more subtrees *)
    BEGIN
        FOR i := 1 TO no. of subtrees DO IN PARALLEL
                TREESEARCH(ith child of node);
        IF any of them reported success THEN report success
        ELSE report failure
    END
```

In case a node has subtrees, a strictly sequential approach would be to test the children one by one until a success is reported or all the children nodes have been tried. This sequential approach can be written as follows:

```
 REPEAT
        pick an untried child n of the node;
        TREESEARCH(n)
 UNTIL successful or node has no more untried children;
```

In procedure TREESEARCH above, we have shown the parallelism in backtracking by invoking TREESEARCH in parallel for all children of a node.

Consider the search tree generated for the acceptor of a regular language. Given a sentence, the acceptor tries to find a sequence of nodes in the tree starting from the node corresponding to the initial state. If upon reaching the end of the string the acceptor finds itself in a nonfinal state then the string is rejected. From a node in this tree there may be more than one transition on the same alphabet to other nodes.

Choosing a transition arbitrarily and proceeding may either lead to a final state (in which case the string is accepted) or to a nonfinal state from which there are no valid transitions. In the latter case, backtracking is done to one of the previous states and an alternative branch is tried. If from a state there are several alternative branches on the same alphabet then the right branch to take may not be known until later. If the searches on all equally valid branches are done in parallel this backtracking could possibly be avoided. This would require effective communication between the parallel tasks so that when one of the states reached is a final state the other tasks could be informed so they could suspend searching.

The search technique in the tree in this case involves an 'OR' logic whereby a state reports success if any of its children does.

In general, nondeterministic computations have search trees with an exponential number of nodes (as in the case of game trees, etc.), making it impractical to take all

possible transitions from a node either sequentiallly or in parallel. In the latter case, they may require exponential number of processors. Such problems are treated by a good blending of heuristics (such as branch and bound methods) and parallelism. Kung [37] remarks that algorithms that generate a large number of solution candidates from which the true solutions have to be selected have a tree structure but may need an exponential number of processors.

CHAPTER 4

PARALLELISM IN COMPUTING MAXIMAL COMMON SUBSEQUENCES

4.1 MAXIMAL COMMON SUBSEQUENCE ALGORITHM

The following definitions and notations are from Hirschberg [21].

Definitions

Let $A=a_1a_2\ldots a_m$ and $B=b_1b_2\ldots b_n$ be two strings with $m\geq n$. A string $C=c_1c_2\ldots c_p$ is a subsequence of A if and only if there is a mapping $F: \{1,2,\ldots,p\}\rightarrow\{1,2,\ldots,m\}$ such that $F(i)=k$ only if c_i is a_k and F is a monotone strictly increasing function (i.e., $F(i)=u$, $F(j)=v$ and $i<j\Rightarrow u<v$). C is a common subsequence of A and B if it is a subsequence of A and a subsequence of B. It is a maximal common subsequence (MCS) of A and B if it is a common subsequence for which p is maximized.

Notations

Let $D=d_1d_2\ldots d_r$ be a string. By D_{kt} we mean $d_kd_{k+1}\ldots d_t$ if $k\leq t$ and $d_kd_{k-1}\ldots d_t$ if $t\leq k$. When $k>t$, the substring is written as $\hat{D}_{kt}$ to indicate that it is a reverse substring of D. By L(i,j) we mean the maximum length possible of any common subsequence of the substrings $a_1a_2\ldots a_i$ of A and $b_1b_2\ldots b_j$ of B. $L^*(i,j)$ stands for the maximum length possible of any common subsequence of $a_{i+1}\ldots a_m$ and $b_{j+1}\ldots b_n$.

Many algorithms have been proposed for recovering a MCS of A and B. Hirschberg [21] presents a quadratic time and linear space algorithm for this using a divide and conquer approach. It is based on the following result:

Given two strings A of length m and B of length n, it is possible to find substrings A_1, A_2 of A and B_1, B_2 of B such that

(1) $A = A_1 || A_2$ and $B = B_1 || B_2$

(2) $MCS(A,B) = MCS(A_1, B_1) || MCS(A_2, B_2)$

where $||$ stands for the catenation operation.

The algorithm appropriately divides each of the strings A and B into two parts, recovers the MCS of the substrings and combines them to get the MCS of A and B. This generates a binary tree structured algorithm as the problem of finding a MCS between two strings is broken into two problems of finding a MCS of two pairs of substrings in a recursive manner.

The algorithm requires that the strings A and B are divided appropriately so that the above result is true. The procedure for this division is as follows:

(1) Choose A_1 to be the substring $a_1 \cdots a_{m/2}$ and A_2 to be the substring $a_{m/2+1} \cdots a_m$ (Here m/2 stands for $\lfloor m/2 \rfloor$).

(2) Compute $\max_{1 \leq j \leq n} \{ L(m/2, j), L^*(m/2, j) \}$. Let m_1 be the minimum j for which this maximum occurs.

(3) Choose B_1 to be the substring $b_1 \cdots b_{m_1}$ and B_2 to be the substring $b_{m_1+1} \cdots b_n$.

The quadratic time and space algorithm given in Wagner and Fischer [50] has been modified by Hirschberg [21] to a quadratic time and linear space algorithm which constructs a vector LL (where LL(j) corresponds to the maximum length possible of any common subsequence of the string A and the substring $b_1 \ldots b_j$ of B). In the next section we discuss a parallel realization of these algorithms.

Referring to the algorithm for computing the vector LL as LEN we can write the parallel algorithm for getting a MCS of A and B as shown below.

```
MCS(m,n,A,B,C);
    IF n=0 THEN C :=  empty string
    ELSE IF m=1 THEN IF ∃ j≤n such that A(1)=B(j)
                              THEN C := A(1)
                         ELSE C := empty string
         ELSE (* m>1, n>0 *)
           BEGIN i := ⌊m/2⌋;
                  LEN(i,n,A1i,B1n,L1);
                         AND
                  LEN(m-i,n,Âm,i+1,B̂n1,L2);
                  M := max_{0≤j≤n} { L1(j)+L2(n-j) };
                  k := min.j such that L1(j)+L2(n-j)=M;
                  MCS(i,k,A1i,B1k,C1);
                         AND
                  MCS(m-i,n-k,Ai+1,m,Bk+1,n,C2);
                  C := C1 || C2
           END
END MCS.
```

Note

The connective AND is used in the above algorithm to denote that the two invocations can be done in parallel.

As the two invocations of LEN and the two invocations of MCS operate on independent substrings of the strings A and B, the correctness of MCS follows from that of ALG C given by Hirschberg [21].

Example

Let A be the string 'ababacbab' and B be the string 'aabcaba'. Then the binary tree generated by algorithm MCS will be as in Figure 4.1. The symbol 'Δ' is used in the figure to denote a blank character. Each node in this tree is labelled in the form $(x,y) \rightarrow z$, where x and y are the substrings of A and B respectively and z is an MCS of x and y. The number of levels in this tree is 5 and the number of nodes is 16 corresponding to the 16 invocations of algorithm MCS. The number of leaves in the tree is 9 which indicates the maximum number of processors that can be used in parallel.

4.1.1 Analysis of MCS

As noted earlier, MCS has a binary tree structure as it is invoked twice recursively in every invocation. The number of levels in the binary tree depends upon m and n where m_1 and m_2 are the lengths of the substrings of A such that $m=m_1+m_2$. A node in this tree represents an action which corresponds to an invocation of MCS. Each invocation

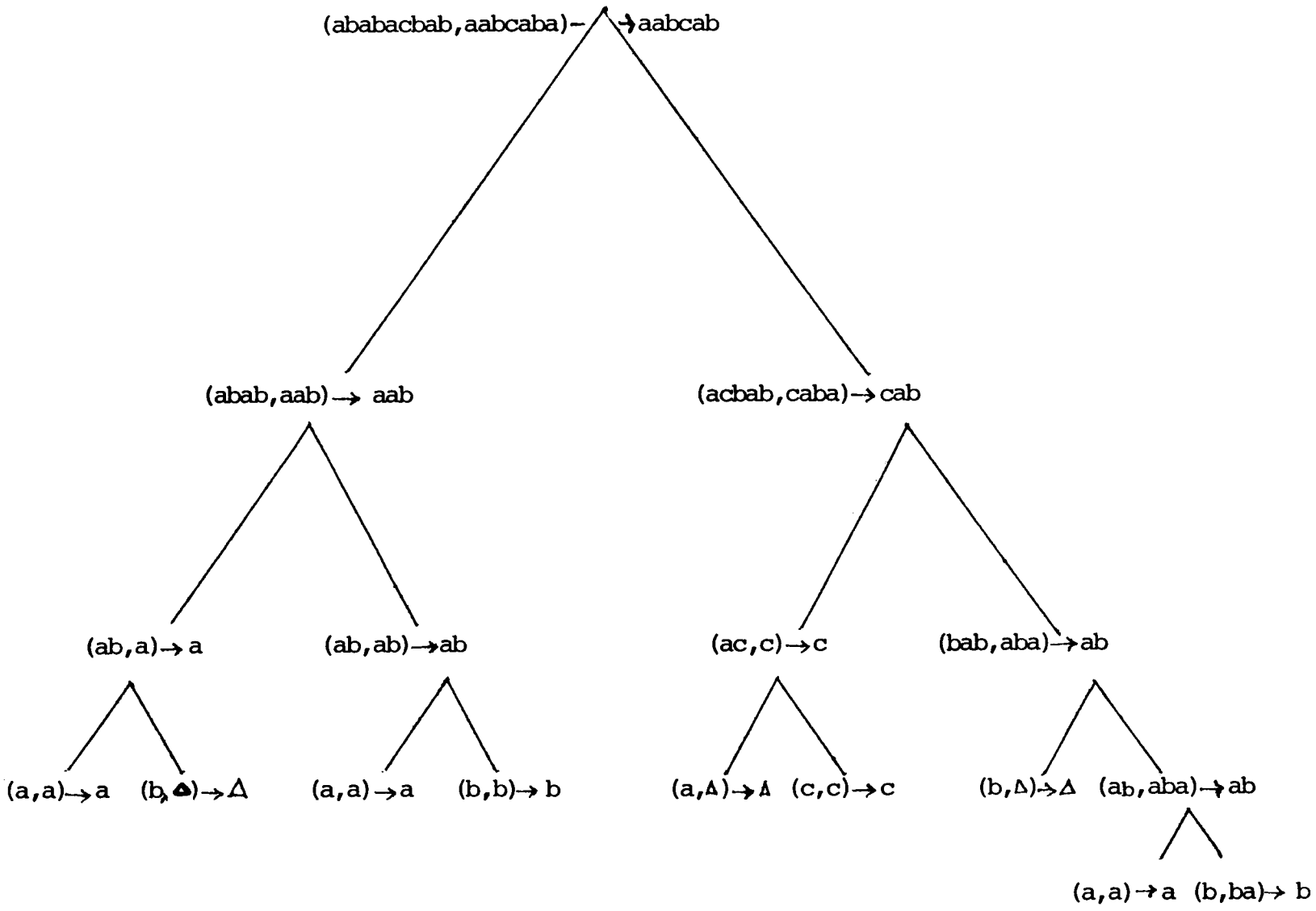

Figure 4.1 Tree Structure of MCS Algorithm.

of MCS involves two calls to LEN and O(m) operations on vectors apart from two calls to itself. The two invocations of LEN can themselves be done in parallel as they operate on independent substrings.

The flow of computation is in the forward direction as calls to MCS are generated with smaller and smaller length substrings. When the leaves of the tree are generated with substrings of length 1, the flow of computation is backward to combine the solutions. This algorithm can thus be classified as a synchronous tree algorithm.

The number of invocations of MCS has been shown by Hirschberg [21] to be bounded by (2m-1). Let $m=2^r$ for some integer $r \geq 0$. When r=0, there is one call to MCS. When r>0, there are two calls to MCS with lengths of substrings m_1 and m_2 such that $m_1+m_2=m$ and both m_1 and m_2 are at most 2^{r-1}. Hence, the number of invocations is given by the recurrence

$$I(m) = 2I(m/2) + O(1)$$

$$I(1) = 1$$

where I(m) is the number of invocations with string length m. The solution of this recurrence is (2m-1). The degree of parallelism of this tree is m. With all invocations corresponding to the nodes at a level in the tree done in parallel the number of invocations is $(\log_2 m+1)$ with at most m processors. The speed-up is thus $O(m/\log_2 m)$. The time bound for LEN is shown by Hirschberg [21] to be $O(m^2)$. See also section 4.2 for a discussion on algorithm LEN and its complexity. So, the order relation for the time taken

by algorithm MCS is given by,

$$T_{mcs}(m) = 2\,T_{mcs}(m/2) + 2\,O((m/2)^2) + O(m) \text{ with}$$

$$T_{mcs}(1) = O(1)$$

whose solution is given by $O(m^2 + m \log_2 m)$.

With two processors available at every level of the tree the order relation can be written as,

$$T_{mcs}(m) = T_{mcs}(m/2) + O(m/2) + O(m)$$

$$T_{mcs}(1) = O(1)$$

using the fact that LEN can also be executed in parallel and that it takes O(m) time in parallel.

The solution to this recurrence is 2m-1 which gives the time bound for MCS in parallel with atmost m processors.

4.2 COMPUTING THE VECTOR LL

Consider the algorithm given below to compute the L matrix associated with the strings A and B:

```
ALG A(m,n,A,B,L);
    1. Initialization: L(i,0) := 0  for i=0 to m;
                       L(0,j) := 0  for j=0 to n;
    2. FOR i := 1 to m DO
           BEGIN
               FOR j := 1 to n DO
                   IF A(i)=B(j) THEN L(i,j) := L(i-1,j-1)+1
                   ELSE L(i,j) := max L(i,j-1), L(i-1,j)
           END
END ALG A.
```

Algorithm A accepts as input two strings A_{1m} and B_{1n} and computes the matrix L where L(i,j) is the maximum length possible of any common subsequence of A_{1i} and B_{1j}. Notice that an element L(i,j) can be computed as soon as L(i-1,j-1), L(i,j-1) and L(i-1,j) are known. Representing this dependency as in Figure 4.2, we can draw dependency graphs for computing the elements of L given m and n.

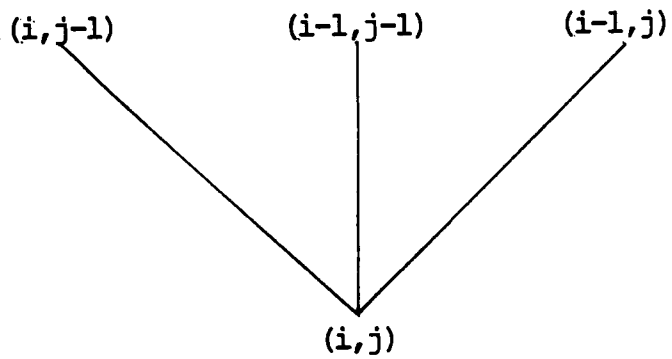

Figure 4.2 Dependency Relation.

We show in Figures 4.3 and 4.4 two example dependency graphs. The nodes in the graphs are labelled as a pair indicating the row and column positions in the L matrix respectively. Note that nodes with a 0 either in the first or second position do not depend on any other nodes.

We assign level numbers to the graph as shown so that at level k the elements L(i,j) such that (i+j-1)=k are computed. The computations at a given level can go in parallel. There are (m+n+1) levels in the graph.

An algorithm to compute the matrix L exploiting the parallelism in computing the elements in a level would be as follows:

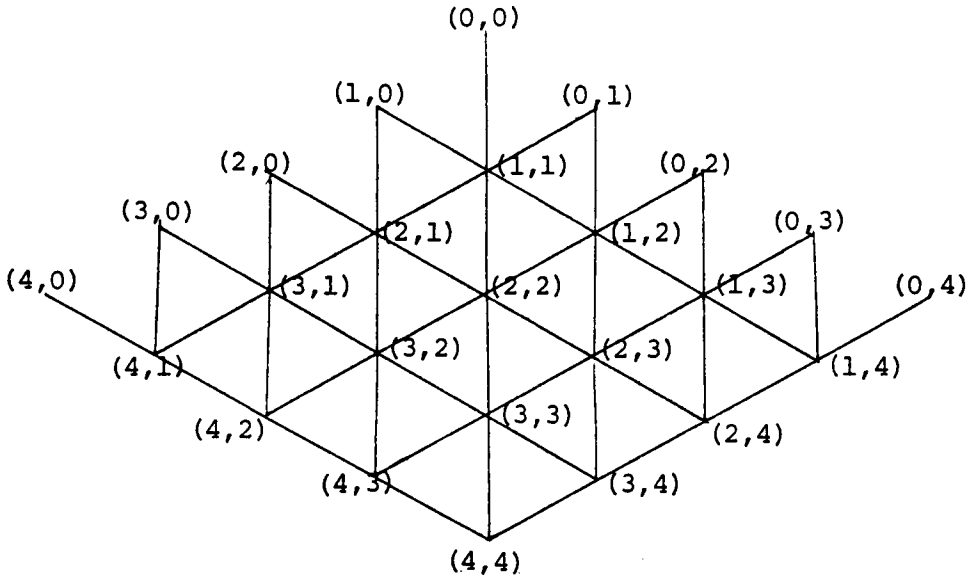

Figure 4.3 Dependency Graph for m=n=4.

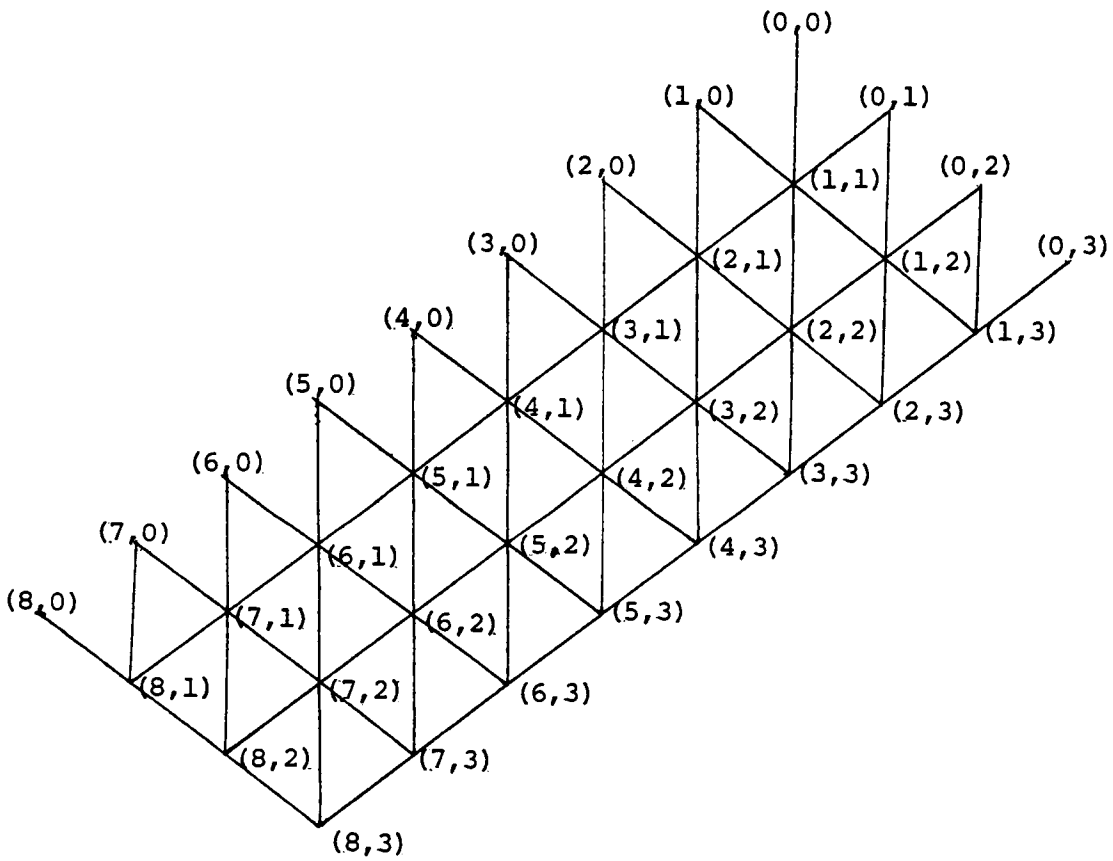

Figure 4.4 Dependency Graph for m=8, n=3.

```
ALG AP(m,n,A,B,L);
 1. Initialization: L(i,0) := 0  FOR i=0 to m;
                    L(0,j) := 0  FOR j=0 to n;
 2. FOR j := 1 TO n DO IN PARALLEL;
      FOR i := 2-j TO m DO
          IF i<0 THEN L(i,j) := 0
          ELSE
            IF A(i)=B(j) THEN
                L(i,j) := L(i-1,j-1)+1
            ELSE
                L(i,j) := Max  L(i,j-1),L(i-1,j) ;
END ALG AP.
```

Let the parallel computations be done in n processors P(1),P(2),...,P(n). In processor P(j), elements L(i,j) for i=(2-j) to m are computed in sequence. The computation of L(i,j) in P(j) requires the values of L(i-1,j-1) and L(i,j-1) computed by P(j-1) and the value of L(i-1,j) computed by P(j) itself. Thus it is important that the computations in different processors are properly synchronized as follows: Let it take unit time for computing an element of L(i,j). Then in all processors the computation of the first element (corresponding to i=2-j) starts at t=0, the second element (corresponding to i=2-j+1) starts at t=1, etc., where t denotes time. That is, the computation of the i-th element starts at t=(i-1) strictly succeeding the computation of the (i-1)th element starting at t=(i-2).

For instance, the starting time for computing L(i,j) for m=n=4 is presented in Table 1. The table shows the time t and the position (i,j) in the matrix L computed during that time.

Analysis of ALG AP

ALG AP computes the matrix L in levels where at level k all elements L(i,j) such that (i+j-1)=k are computed. There are (m+n+1) levels in all. For instance, when m=8 and n=3 the matrix L is computed in the order shown in Table 2 where the numbers in a position refers to the level number at which the corresponding element is computed. This is the difference between ALG AP (where computations are done along levels) and ALG A (where computations are done row wise).

ALG A requires O(mn) space and O(mn) time; see Hirschberg [21]. In ALG AP, the maximum no. of computations which can go in parallel is n. So, if there are n processors, the maximum computation occurs in P(n) and it is of order O(m+n-1). The space requirement remains the same as ALG A. This does not take into account the broadcast of the program into n processors initially which takes about $O(\log_2 n)$. If there are p processors then the algorithm would alter in the beginning as follows:

```
FOR k := 1 TO ⌈n/p⌉ DO
   FOR j := 1 TO p DO IN PARALLEL
     FOR i := (2-j) TO m DO
           ...
```

TABLE 1

Time t	processor 1	processor 2	processor 3	processor 4
0	(1,1)	(0,2)	(-1,3)	(-2,4)
1	(2,1)	(1,2)	(0,3)	(-1,4)
2	(3,1)	(2,2)	(1,3)	(0,4)
3	(4,1)	(3,2)	(2,3)	(1,4)
4		(4,2)	(3,3)	(2,4)
5			(4,3)	(3,4)
6				(4,4)

TABLE 2

	Col 1	Col 2	Col 3
Row 1	1	2	3
Row 2	2	3	4
Row 3	3	4	5
Row 4	4	5	6
Row 5	5	6	7
Row 6	6	7	8
Row 7	7	8	9
Row 8	8	9	10

The complexity becomes n/p O(m+n-1) with p processors.

There is no interaction between the computations which are going on in parallel. Even the contention to a common memory location does not pose any serious delay as any particular L(i,j) is used only by 3 processors. As the computation proceeds in levels with sequential execution among levels and parallel execution within a level there are only 2 processors trying to access a common location. So, the communication delay comes from switching from one level to another. This switching takes place in the processor itself and takes constant time. Thus if the computations are initially stored in n processors then the total complexity is O(2n,n). Storing computation in n processors takes $O(\log_2 n)$.

The independent tasks in ALG AP are of low module granularity. This makes it suitable for implementation as a systolic algorithm. The design for this is derived from the dependency structure of this algorithm shown earlier. See the dependency graph (Figure 4.2) for m=n=4.

Consider a network of processing elements each of which can compute an element L(i,j) of the matrix L. The input to each processor consists of 5 items: L(i-1,j-1), L(i,j-1), L(i-1,j), a_i and b_j. The output, L(i,j), feeds 3 other processing elements. Thus, assuming a suitable shape for each processing element the underlying geometry of the interconnections between processors can be worked out. For

instance, for m=n=4, the inconnection structure is as shown in Figure 4.5. Here a square represents a processing element. Assuming unit time for computing by each processor and unit time for propagation of results from one processor to another, the computation time is bounded by the number of levels which is (m+n-1) and the communication time is bounded by (m+n-2).

4.2.1 Parallel Linear Space Algorithm - ALG LEN

Noting that the goal is to compute L(m,n) and in ALG A computation of the ith row requires knowledge of only (i-1)th row elements, Hirschberg [21] presents a linear space algorithm to compute a vector LL instead of the matrix L, where LL(j) is the maximum length possible of any common subsequence of the string A and the substring B_{1j} of B. The algorithm is shown below:

```
ALG B(m,n,A,B,LL);
  1. Initialization: K(1,j) := 0  FOR j=0 TO n;
  2. FOR i := 1 TO m DO
        BEGIN
          K(0,j) := K(1,j)  FOR j=0 TO n;
          FOR j := 1 TO n DO
             IF A(i)=B(j) THEN K(1,j) := K(0,j-1)+1
             ELSE K(1,j) := max K(1,j-1),K(0,j)
        END
  3. LL(j) := K(1,j)  FOR j=0 TO n;
END ALG B.
```

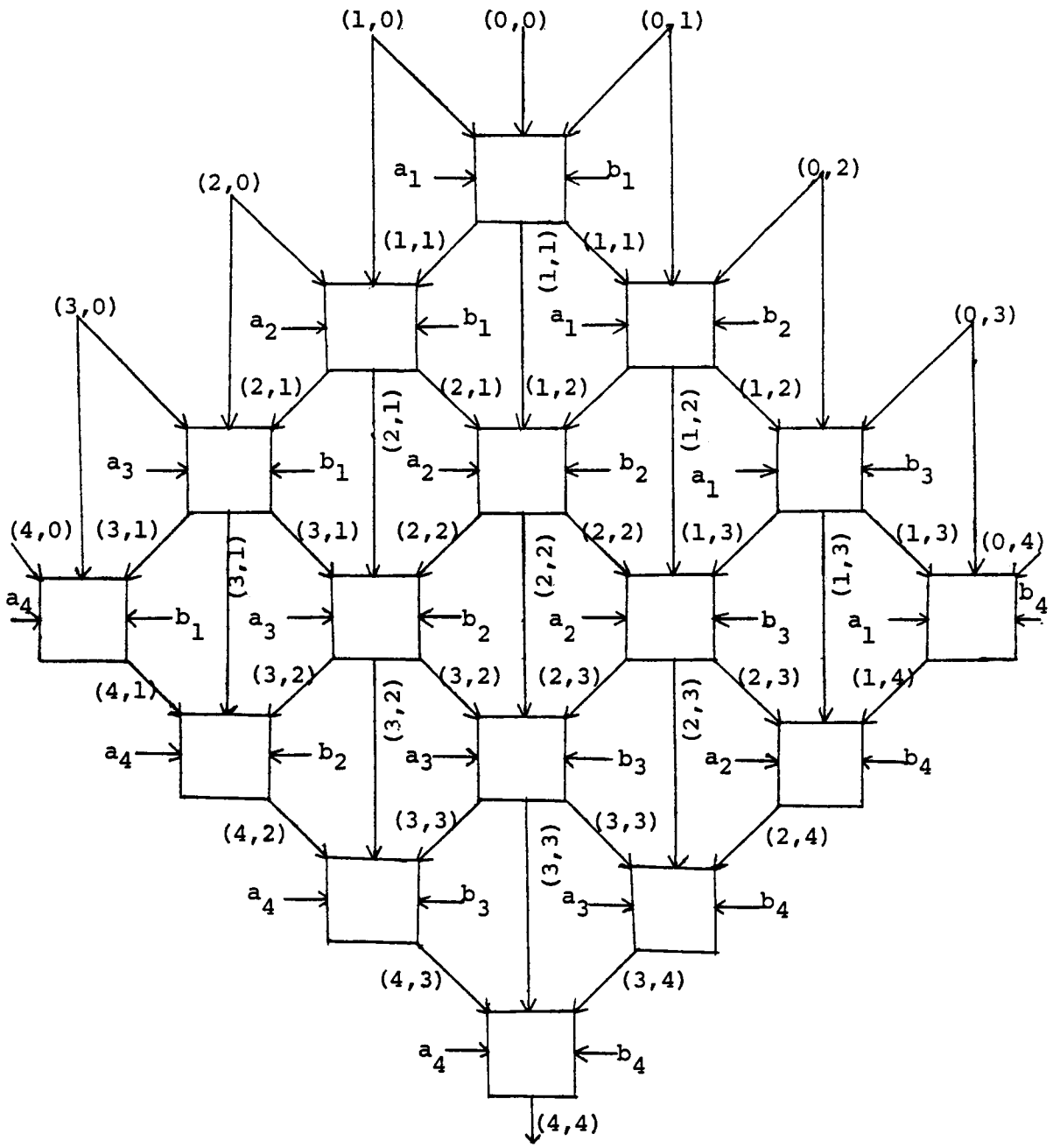

Figure 4.5 Systolic Array for m=n=4.

A similar improvement to ALG AP also produces a linear space parallel algorithm. Like ALG B, a vector LL is produced containing the maximum lengths possible of any common subsequences of A and B for j=1 to n. The algorithm is such that the computations in a processor P(j) uses the three values K0(j), K1(j) and K1(j+1).

```
LEN(m,n,A,B,LL);
  1. Initialization:
        FOR j := 1 TO n DO
            BEGIN
               K0(j) := 0; K1(j) := 0; K2(j) := 0
            END
  2. FOR  j := 1 TO n DO IN PARALLEL
           FOR i := (2-j) TO m DO
              BEGIN
                IF i≤0 THEN  LL(j) := 0
                ELSE
                    IF  A(i)=B(j) THEN
                          LL(j) := K0(j)+1
                    ELSE
                          LL(j) := max  K1(j),K1(j+1)  ;
                K0(j) := K1(j);
                K1(j+1) := LL(j)
              END
END LEN.
```

Note

The contention to common storage present in ALG AP has been carried over to LEN since the K1 value used by processor P(j+1) is used and modified by processor P(j). For instance, in the parallel tasks corresponding to j=1 and j=2, K1(2) is required by both the tasks. The accesses should be so synchronized that the modifications made in a time period t_i are not effective until the next time period t_{i+1}.

CHAPTER 5

A PARALLEL ALGORITHM FOR COMPUTING POLYNOMIAL ZEROS

5.1 THE ALGORITHM

A global bisection algorithm was suggested by Wilf [52] for computing all the zeros of a polynomial P(Z) with complex coefficients. Recently, Krishnamurthy and Venkateswaran [32] presented a parallel version of Wilf's algorithm. For full details about the mathematical basis and the algorithms, see these two references.

Wilf's algorithm is based on the ability to count the number of zeros of a complex polynomial contained within a given rectangle in the complex plane. Then starting with a rectangle containing all the zeros of P(Z) one can repeatedly bisect this rectangle into smaller ones and count the zeros within. This can be done until a required tolerance is achieved. Parallelism is possible in this algorithm as the counting of the number of zeros in different subrectangles can go on independently.

A recursive parallel algorithm for computing the zeros of a given polynomial p(z) is presented below. We have used the connective AND to denote that the two invocations can be done in parallel.

```
PROCEDURE BISECTION(R,eps,n);
    BEGIN
        IF side of R > eps THEN
           BEGIN
               Subdivide R as Left/Right or Top/Bottom
               appropriately, say R1 and R2;
               Count the number of zeros NR1 in R1;
               Set the no. of zeros NR2 in R2 to n - NR1;
               IF R1 has n zeros THEN
                           BISECTION(R1,eps,NR1)
               ELSE IF R2 has n zeros THEN
                           BISECTION(R2,eps,NR2)
                    ELSE
                           BISECTION(R1,eps,NR1)
                                 AND
                           BISECTION(R2,eps,NR2)
           END
        ELSE the n zeros := the centre of R
    END
END BISECTION.
```

Since the computational effort for locating a zero in a rectangle is greater than that for locating a zero on a line, we assume the worst case where no zeros lie on a division line. The procedure is invoked initially with a square S containing all the zeros of P(Z).

5.2 STRUCTURE OF THE ALGORITHM

This algorithm has a binary tree structure as a bisection step generates two bisection steps which are invoked recursively. The possibility of dividing the data space into two independent subsections and applying PROCEDURE BISECTION on each separately lends parallelism to the algorithm.

Let the size of the initial square S be 2^m (where $m \geq 0$) and the tolerance be 2^{-e}. This means that finally the zeros are to be bounded by squares of size 2^{-e}. Let $p = m + e$. Then the binary tree has 2p levels because a vertical subdivision of a square produces a rectangle and a horizontal subdivision of this rectangle produces a square half the size of the original square. In a complete binary tree with 2p levels which are numbered from 0 to (2p - 1) the following are true; see Knuth [27]:

(a) There are 2^j nodes at level j;

(b) 2^{j+1} nodes emanate from the node at level j for $j<2p-1$;

(c) Between a node at level j and a terminal node there are (2p-1-j) nodes excluding the node at level j.

But the tree structure that this algorithm produces need not be a complete binary tree. If there is only one zero to be located, then the corresponding binary tree would have only one branch emanating from the node at any level j, for j=0 to (2p-1). If there is one zero in the left half

and another in the right half of the square S, then the binary tree has two branches emanating from the node at level 0. There will be only one branch emanating from the nodes at all other levels. In the case of three zeros with wide distribution in the square S, the tree would have two branches at level 0, two branches from one of the two nodes at level 1 and a single branch from all other nodes.

Example

Let m=0 so that the size of the initial square S is 1. Let the tolerance = 2^{-3}. Then p = 3 and hence 2p = 6.

For n=1,2,3,4 respectively, the binary trees are as shown in Figure 5.1.

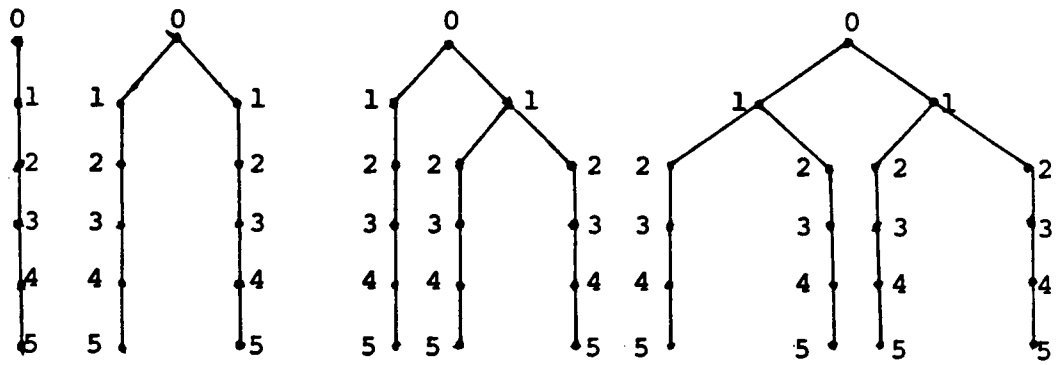

Figure 5.1 Binary Trees for n=1,2,3,4.

Since the operations being performed on independent data items (namely, the subrectangles) is the same (namely, PROCEDURE BISECTION), the algorithm can be classified as a SIMD algorithm. But the algorithm is suitable for

asynchronous implementation for the following reasons:

. the flow of computation proceeds in one direction (from the root of the binary tree to its terminal nodes) without requiring synchronization among its independent tasks;

. the only communication between processors is when a processor wants to hand over computation (PROCEDURE BISECTION) to two other processors;

. the amount of time for computation in each processor exceeds by a factor of n, the amount of time for a communication step (see below).

As noted above, the flow in this algorithm is in the forward direction, thus making it an asynchronous tree algorithm. The number of leaves in the tree is n when all the n zeros of the polynomial are distinct with respect to a specified tolerance 2^{-e}. The number of leaves n is also the maximum degree of parallelism in the algorithm.

5.3 COMPLEXITY

The computational complexity of this algorithm with n processors is discussed by Krishnamurthy and Venkateswaran [32]. It is shown to be $O(n^2p)$ without considering communication complexity. Here we show that the communication takes O(np) time so that the effective complexity of the parallel algorithm is still $O(n^2p)$. We assume that there are n processors P(1),P(2),...,P(n) with their own local memory which can communicate with each

other. At every level, when a processor P(i) has to choose 2 processors to hand over the counting in two subrectangles, it can choose one available processor for doing the computation using one subrectangle and P(i) itself can do the computation in the other subrectangle. There are several topological interconnection structures for these processors to realize the tree structure of the algorithm. For instance, a processor P(i) can be linked to two other processors P(2i),P(2i+1) as a tree machine. Thus, it is reasonable to assume that it takes constant time to transmit a fixed size data from a processor to another at any level of computation. We further assume that access to common data items in storage by different processors is avoided by having all data items required for counting in a subrectangle in the local memory of a processor itself. Thus the communication between 2 processors requires the transmission of the following items of data:

Size of the rectangle to be searched.

The coordinates of the rectangle.

Number of zeros to be searched in the rectangle.

The tolerance.

The searching procedure (PROCEDURE BISECTION).

The polynomial itself (say, as a coefficient vector).

All data items except the last one are of fixed size so that the time taken for transmitting these items is a fixed constant, say Tc. Hence, one communication step takes

(O(n)+Tc) time, where O(n) is the time taken to transmit the coefficients of the polynomial from one processor's local memory to another processor's local memory. Now, the flow of computation being only in one direction, at every level of the binary tree there is one communication step going on in parallel in several processors. As there are 2p levels in the binary tree, the total communication time is given by 2p (O(n) + Tc) which is equivalent to O(np).

Using the results of Krishnamurthy and Venkateswaran [32], we can estimate, as follows, the efficiency gain in going from a sequential to parallel execution in the general case when the n zeros of the polynomial are widely distributed in the initial square S.

The number of nodes in the binary tree in this case is O(np). An action represented by a node in the binary tree is the execution of PROCEDURE BISECTION which takes $O(n^2)$ time, so that the total time for sequential execution is $O(n^3p)$. With n processors the total time for parallel execution is $O(n^2p)$.

Thus, the efficiency gain is given by (O(np) - O(p))/O(np) which is equivalent to O(1). The speed-up is given by $O(n^3p)/O(n^2p)$ which is equivalent to O(n).

CHAPTER 6

DYNAMIC PROGRAMMING APPLIED TO A MAXIMIZATION PROBLEM

6.1 INTRODUCTION

In this chapter, we consider the parallelization of a dynamic programming (DP) algorithm for a maximization problem. The discussion is based on the paper by Gilmore [19] in which it was pointed out how a binary tree structured algorithm can be given by a proper rearrangement of the stages in the DP approach.

Consider the resource allocation problem:

Let x be a resource which has to be allocated to some n activities. Let x_i be the allocation to each activity so that $\sum_{1}^{n} x_i = x$. Let $g_i(x_i)$ be the resultant return from activity i.

The problem is to find out an allocation policy such that the total return $\sum_{1}^{n} g_i(x_i)$ is maximized.

This is an instance of the more general problem of maximizing a function of n variables,

$R_n(x_1, x_2, \ldots, x_n) = \sum_{1}^{n} g_i(x_i)$ over the region

$S_n(x) = \left\{(x_1, x_2, \ldots, x_n) \;\middle|\; \sum_{1}^{n} x_i = x,\ x_i \geq 0\right\}$ with $g_i(0) = 0$ and $g_i(x_i) \geq 0$.

The techniques of Dynamic Programming are applied to solve this problem. It is based on the optimality principle which states that an optimal policy has the property that

whatever the initial state and initial decision are, the remaining decision must constitute an optimal policy with regard to the state resulting from the first decision.

Translated in terms of the resource allocation problem this means that given a partial allocation policy which is optimum upto a certain stage, say $g_i(x_i)$, an optimal allocation policy for all the activities can be got by finding an optimum allocation policy for the remaining activities $g_{i+1},\ldots,g_n$.

The technique is to construct the solution in stages, every stage using the partial solution obtained up to that stage. This is usually realized in sequential fashion. But by rearranging the stages it is possible to inject parallelism in the solution procedure as discussed by Gilmore [19] in one of the early papers on parallel algorithms.

6.2 SEQUENTIAL EXECUTION

To maximize $R_n(x_1,\ldots,x_n)$ a sequence of functions $f_1(x), f_2(x),\ldots,f_n(x)$ is constructed where $f_k(x)$ is defined as follows:

$$f_k(x) = \max_{S_k(x)} [R_k(x_1,\ldots,x_k)] = \max_{0 \leq x_k \leq x} [g_k(x_k) + f_{k-1}(x-x_k)]$$

and $f_0(x) = 0$

Corresponding to each function $f_k(x)$ a function $x_k(x)$ is also constructed where $x_k(x)$ is the allocation to $g_k(x_k)$ which maximized $f_k(x)$.

Discretizing the range the result is a table of 2n columns, the ith pair of columns $f_i(j)$ giving the function values f_i and $x_i(j)$ giving corresponding x values. This represents the optimum policy up to the i-th stage.

Given a resource x, the allocation $x_1, x_2, \ldots, x_n$ corresponding to the functions $g_1, g_2, \ldots, g_n$ which maximizes $\sum_{1}^{n} g_i(x_i)$ is read out from the table.

6.2.1 Reading Out from the Table

Let x be a given resource. Let i be the index of x with respect to the discretization. (For instance, if the original range is [0,1] and it is divided into 11 points with an interval of 0.1, then x = 0.6 has index i = ((0.6-0)/0.1) + 1 = 7). Then $x_n(i)$ from the table gives the allocation to $g_n(x)$. Then $[x-x_n(i)]$ is the resource available for the functions $g_1, g_2, \ldots, g_{n-1}$. Let j be the index of $[x-x_n(i)]$. Then $x_{n-1}(j)$ is the allocation to $g_{n-1}(x)$ and so on. A procedure for this can be written as follows:

```
PROCEDURE READOUT;
  (* Input:x, the resource; Output:the allocation policy *)
     beta := x; j := index of x;
     FOR k := n DOWNTO 1 DO
          BEGIN  alpha := x(k,j)
              PRINT f(k,j); PRINT k,alpha ;
              beta := beta - alpha; j := index of beta
          END
END READOUT.
```

6.2.2 Complexity of Sequential Execution

We now derive an upper bound for the time required for the maximization algorithm in the sequential execution case. This is based on the algorithm (procedure MAXIMIZE, see section 6.3.1) to maximize two given functions.

Let the range of x be divided into (m-1) divisions (m points), $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$. $f_1(x^{(i)}) = g_1(x^{(i)})$ requires m evaluations of the function g_1. For k>1, $f_k(x^{(i)})$ is constructed in 2 stages.

In stage 1, an intermediate column of function values $f_k(x^{(i)})$ is produced where $f_k(x^{(i)}) = \max_{1 \leq j \leq i} (f_{k-1}(x^{(j)}))$ for i=1 to m.

In stage 2, $f_k(x^{(i)})$ is computed as,

$f_k(x^{(i)}) = \max_{1 \leq j \leq i} (g_k(x^{(j)}) + f_k(x^{(j)}))$ for i=1 to m.

In stage 1, $0+1+2+\ldots+(m-1) = (m(m-1))/2$ operations (comparisons) are done.

In stage 2, $(1+2+\ldots+m)$ additions and $0+1+2+\ldots+(m-1)$ comparisons are done given the m function values $g_k(x^{(j)})$ for j=1 to m. Hence the complexity of constructing the values of the function f_k using the values of the function f_{k-1} requires $O(m^2)$ operations where an operation can be a comparison or an addition. This is the time taken to maximize two functions. The construction of the values of the functions $f_1, f_2, \ldots, f_n$ in sequence thus requires a total of $(n-1)O(m^2) = O(nm^2)$ operations. This does not include the evaluation of the n functions $g_1, g_2, \ldots, g_n$ at the m points $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$. Denoting the time for a single

evaluation by U, the total time for evaluation is O(nm)U. This gives the sequential execution time as $O(nm^2) + O(nm)U$.

6.3 PARALLEL EXECUTION

A reformulation of the maximization problem by suitably grouping the functions g provides the technique for realizing the DP solution in parallel.

Let there be 2n variables $x_1, x_2, \ldots, x_{2n}$ such that we need to maximize

$$R_{2n}(x_1, x_2, \ldots, x_{2n}) = \sum_{1}^{2n} g_i(x_i) \text{ with } \sum_{1}^{n} x_i = x \text{ over}$$

$$S_{2n}(x) = \left\{ (x_1, x_2, \ldots, x_{2n}) \,\middle|\, \sum_{1}^{2n} x_i = x \text{ and } x_i \geq 0 \right\}.$$

Let $x_1 + x_2 + \ldots + x_n = y_1$ and $x_{n+1} + x_{n+2} + \ldots + x_{2n} = y_2$ so that $y_1 + y_2 = x$.

We define $S_1(y_1) = \left\{ (x_1, x_2, \ldots, x_n) \,\middle|\, \sum_{1}^{n} x_i = y_1 \text{ and } x_i \geq 0 \right\}$ and $S_2(y_2) = \left\{ (x_{n+1}, x_{n+2}, \ldots, x_{2n}) \,\middle|\, \sum_{n+1}^{2n} x_i = y_2 \text{ and } x_i \geq 0 \right\}$.

Suppose we maximize the functions

$$U_n(y_1) = \max_{S_1(y_1)} [g_1(x_1) + g_2(x_2) + \ldots + g_n(x_n)] \quad \text{and}$$

$V_n(y_2) = \max_{S_2(y_2)} [g_{n+1}(x_{n+1}) + \ldots + g_{2n}(x_{2n})]$. Then the original problem reduces to maximizing

$$U_n(y_1) + V_n(y_2) \text{ over } \left\{ (y_1, y_2) \,\middle|\, y_1 + y_2 = x \text{ and } y_1, y_2 \geq 0 \right\}$$

In effect the problem of maximizing a function of 2n variables has been reduced to two steps:

1. Maximization of 2 functions of n variables each which can go in parallel.

2. Maximization of a function of 2 variables which is the result of step 1 above.

In Figure 6.1 this is represented as a binary tree.

This can be extended to several levels resulting in a binary tree structure for the algorithm.

For instance,

$R_8(x_1,x_2,\ldots,x_8) = \text{Max } [g_1(x_1)+g_2(x_2)+\ldots+g_8(x_8)]$
over $S_8(x) = \left\{(x_1,x_2,\ldots,x_8) \middle| \sum_1^8 x_i = x,\ x_i \geq 0\right\}$,
can be broken up into 2 problems of 4 variables each.

Let $x_1+ x_2+ x_3+ x_4 = y_1$ and $x_5+ x_6+ x_7+ x_8 = y_2$
$U_4(x_1,x_2,x_3,x_4) = \text{Max } [g_1(x_1)+g_2(x_2)+g_3(x_3)+g_4(x_4)]$
over $S_1(y_1) = \left\{(x_1,x_2,x_3,x_4) \middle| x_1+ x_2+ x_3+ x_4 = y_1,\ x_i \geq 0\right\}$
and $V_4(x_5,x_6,x_7,x_8) = \text{Max } [g_5(x_5)+g_6(x_6)+g_7(x_7)+g_8(x_8)]$
over $S_2(y_2) = \left\{(x_5,x_6,x_7,x_8) \middle| x_5+ x_6+ x_7+ x_8 = y_2,\ x_i \geq 0\right\}$

These can be further broken into two problems each of two variables.

Let $x_1+ x_2 = z_1$ and $x_3+ x_4 = z_2$
$UU_2(x_1,x_2) = g_1(x_1) + g_2(x_2)$ over $\left\{(x_1,x_2) \middle| x_1+ x_2 = z_1,\ x_i \geq 0\right\}$
and
$UV_2(x_3,x_4) = g_3(x_3) + g_4(x_4)$ over $\left\{(x_3,x_4) \middle| x_3+ x_4 = z_2,\ x_i \geq 0\right\}$
Let $x_5+ x_6 = s_1$ and $x_7+ x_8 = s_2$
$VU_2(x_5,x_6) = g_5(x_5) + g_6(x_6)$ over $\left\{(x_5,x_6) \middle| x_5+ x_6 = s_1,\ x_i \geq 0\right\}$
and
$VV_2(x_7,x_8) = g_7(x_7) + g_8(x_8)$ over $\left\{(x_7,x_8) \middle| x_7+ x_8 = s_2,\ x_i \geq 0\right\}$

See Figure 6.2.

If the no. of variables n is a power of 2, say $n = 2^s$, then the tree is a complete binary tree. In the case when n is not an exact power of 2, the treatment of all the functions can be done in different ways.

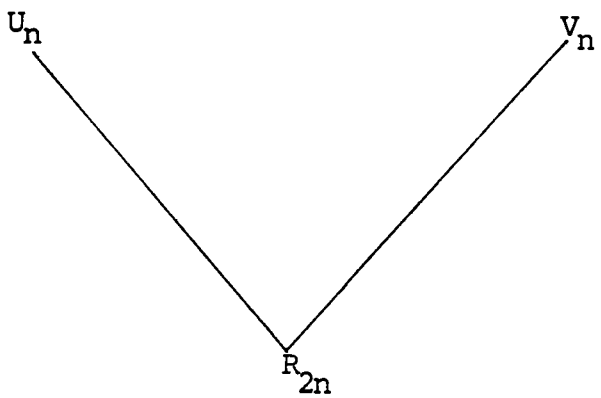

Figure 6.1 Maximization of 2n Variables.

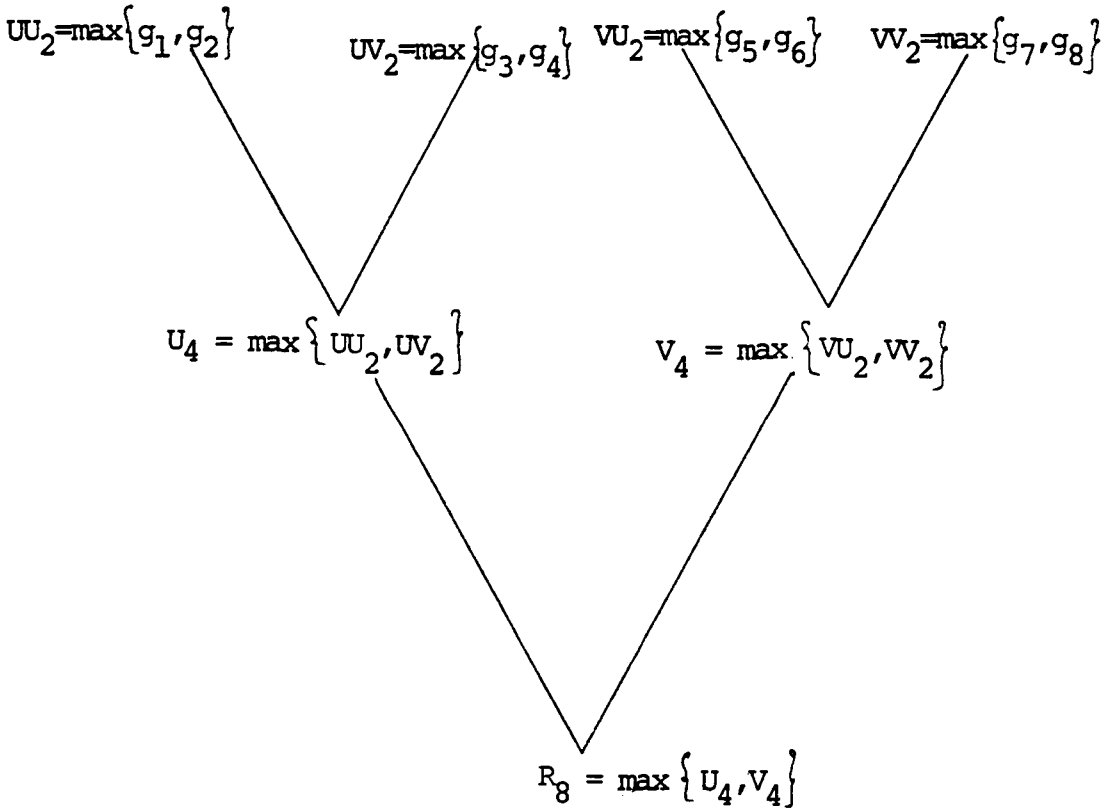

Figure 6.2 Maximization of 8 Variables.

Let there be n functions $g_1, g_2, \ldots, g_n$. Then the functions are paired and maximization performed with (g_1, g_2), $(g_3, g_4), \ldots,$ (g_{n-1}, g_n) if even and (g_1, g_2), $(g_3, g_4), \ldots, (g_{n-2}, g_{n-1})$ if n is odd. This type of pairing is done for resulting functions. Any function that is left out is paired with the final function.

For instance, for n = 6, the tree structure for this approach would be as shown in Figure 6.3.

Another approach would be the following: At every stage, the no. of functions is divided by 2. Any odd function left out is combined immediately. The tree for the case n = 6 is as in Figure 6.4.

6.3.1 Algorithm

Let $g_1, g_2, \ldots, g_n$ be the given functions to be maximized. Let the range x be discretized. Then the dynamic programming algorithm for the maximization problem can be written as follows:

```
PROCEDURE DYNAM(M,p,index);
(* This procedure finds out the maximum function values
for 'p' functions g_M, g_{M+1}, ..., g_{M+p-1}. It uses the 'PROCEDURE
MAXIMIZE' to maximize 2 functions. The resulting function
values are stored in the array 'U' and the corresponding x
values at which the maximum occurs are stored in the array
'Y'. 'index' gives the column no. where the result of
maximizing two functions using 'PROCEDURE MAXIMIZE' should
```

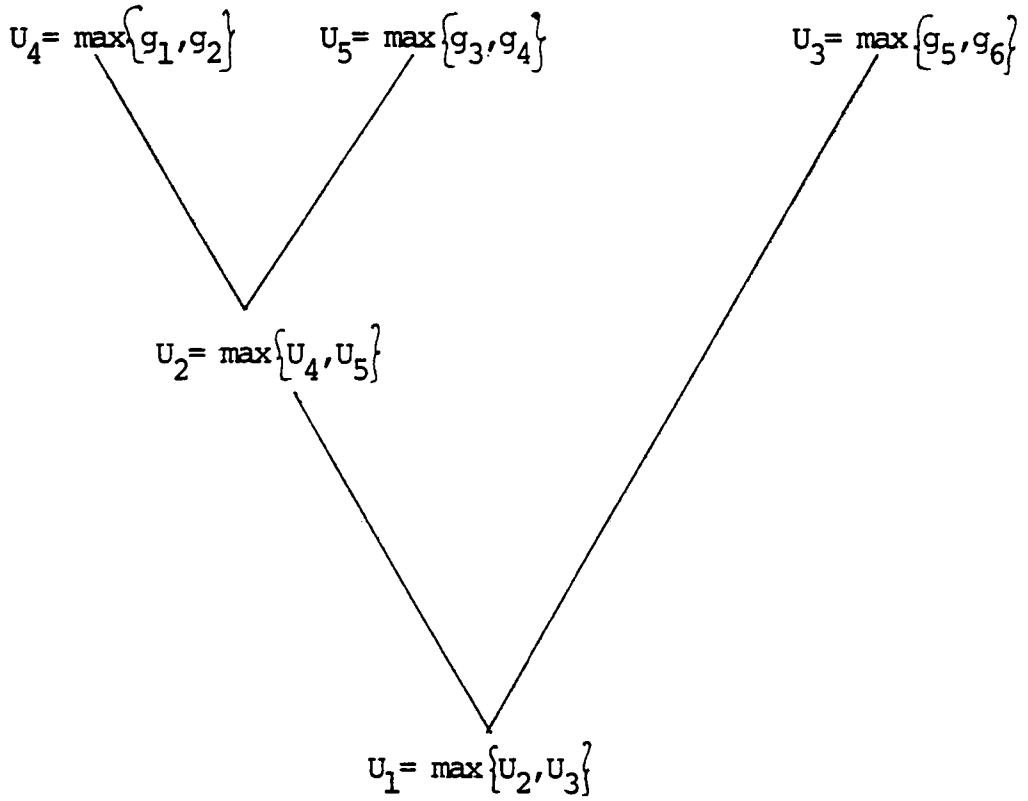

Figure 6.3 Approach 1 for pairing.

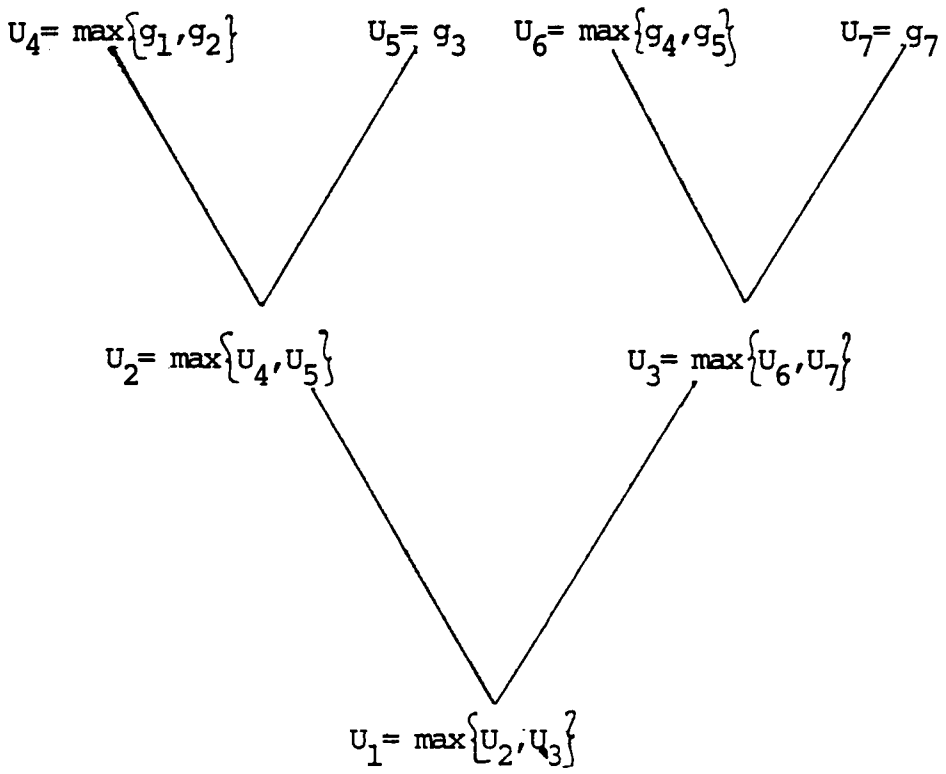

Figure 6.4 Approach 2 for Pairing.

```
be stored.  The discretized x values x_i  (i=1 to N) are known
globally *)
 IF p=1 THEN
          FOR i := 1 TO N DO
               BEGIN U(index,i) := g_M (x(i));
                         Y(index,i) := x(i)
               END
 ELSE IF p=2 THEN
              BEGIN FOR i := 1 TO N DO
                         BEGIN gg(1,i) := g_M (x(i));
                               gg(2,i) := g_M+1 (x(i))
                          END ;
                     MAXIMIZE (gg,index)
              END
     ELSE BEGIN (*There are more than 2 functions.  Split
                  the process into two and then combine*)
                DYNAM(M, p/2 , 2*index);
                          AND
                DYNAM(M+ p/2 , p- p/2 , 2*index + 1);
               FOR i := 1 TO N DO
                         BEGIN
                             gg(1,i) := U(2*index,i);
                             gg(2,i) := U(2*index+1,i)
                         END ;
               MAXIMIZE(gg,index)
          END
END DYNAM
```

Initially, we invoke DYNAM as follows: DYNAM(1,n,1);

```
PROCEDURE MAXIMIZE(gg,index);
    (* This procedure maximizes two functions whose values
are available in gg(1,i), gg(2,i) for i = 1 to N. The
maximum values of the functions are stored in U(index,i) and
the x values where these maxima occur are stored in
Y(index,i) for i = 1 to n. [a,b] is the interval under
consideration which is discretized with a step size h. *)
  BEGIN
   FOR k := 1 TO 2 DO  (*  the max. FOR 2 functions  *)
    BEGIN
    xx := a;   i := 1;
    WHILE xx ≤ b DO   (* use values of xx from a to b *)
     BEGIN
      beta := -1; (* since the function values are > 0,
                  beta is set to a negative value initially
                  to let the first comparison in finding
                  the max. to replace its value *)
      gmold := 0;  j :=1;
      WHILE j ≤ i DO  (* find the x value at which the
                       max.occurs. the max. function
                       value is in beta and the
                      corresponding x value is in gamma *)
       BEGIN
        IF k=1 THEN alpha := gg(1,j)
        ELSE alpha := gg(2,j)+TU(k-1,i-j+1);
```

```
        IF alpha > beta THEN
         BEGIN
          beta := alpha; gamma := gmold
         END;
        j := j+1; gmold := gmold+h
       END;
      TU(k,i):= beta; TY(k,i):= gamma;
      xx := xx+h;  i := i+1
     END;
     FOR i:= 1 TO npts DO
      BEGIN
      U(index,i):= TU(2,i);
      Y(index,i):= TY(2,i)
      END
  END MAXIMIZE;
```

6.3.2 Discussion

The recursive invocation of DYNAM twice in every invocation generates a binary tree structure. An action represented by a node in the binary tree is the maximization of two functions. In all there will be (n-1) such maximizations. The number of levels in the binary tree is $\log n$.

Let $n=2^s$. Then a complete binary tree with s levels is generated. The maximum degree of parallelism in the tree is n/2. The flow of computation is in the backward direction from the leaves to the root. Thus this is an

example of a synchronous tree algorithm. Since the time for performing an action is $O(m^2)$, the time for parallel execution without considering the function evaluations is $O(sm^2)$. The time for function evaluations is 2mU, as n/2 function evaluations are going on in parallel. Thus, the total time for parallel execution is $O(m^2 \log_2 n + O(m)U$.

The efficiency gain is $O(n)/O(\log_2 n)$ since the number of nodes is O(n) and the number of levels is $O(\log_2 n)$.

The binary tree structure of the DYNAM algorithm lends a binary tree structure to the readout procedure. Let the levels in the tree be numbered so that the level of the root node is 0 and the level number of a child node is 1 greater than its parent node. Then the results of the 2^i invocations of DYNAM at level i are stored in the pairs of arrays $(U(2^i), Y(2^i))$, $(U(2^i+1), Y(2^i+1)), \ldots,$ and $(U(2^{i+1}-1), Y(2^{i+1}-1))$, respectively. Here U contains the function values and Y contains the x values at which the maxima occur.

Given a resource x, let j be the index of x with respect to the discretization. Then, Y(1,j) is the allocation to the functions $g_{n/2+1}, g_{n/2+2}, \ldots, g_n$ and x-Y(1,j) is the allocation to the functions $g_1, g_2, \ldots, g_{n/2}$. This is at level 0. At level 1, x-Y(1,j) is used to determine the allocation to the two sets of functions $\{g_1, g_2 \ldots, g_{n/4}\}$ and $\{g_{n/4+1}, g_{n/4+2}, \ldots, g_{n/2}\}$; Y(1,j) is used to determine the allocation to the sets of functions $\{g_{n/2+1}, g_{n/2+2}, \ldots, g_{3n/4}\}$ and $\{g_{3n/4+1}, g_{3n/4+2}, \ldots, g_n\}$. By recursively applying this to

all levels, the READOUT procedure can be written as follows:

```
PROCEDURE READOUT(x,p,index,k,xx);
  BEGIN
    ind := ((x-a)/h ) +1;
    IF p=1 THEN xx(index ) := Y(k,ind)
    ELSE
      IF p=2 THEN
         BEGIN
         xx(index) := x-Y(k,ind); xx(index+1) := Y(k,ind)
         END
      ELSE
         BEGIN
         READOUT(x-Y(k,ind),p div 2,index,2*k,xx);
         READOUT(Y(k,ind),p - (p div 2),index+(p div 2),
                              2*k+1,xx)
         END
END READOUT.
```

The procedure is initially invoked as:

```
READOUT(x,n,1,1,xx);
```

CHAPTER 7

MULTIVARIATE POLYNOMIAL EVALUATION

Algorithm

Let $P(x_1,x_2,\ldots,x_r) = A_0 + A_1 x_r + A_2 x_r^2 + \ldots + A_n x_r^n$ ...(1) be a multivariate polynomial (MVP) in r variables. Here $A_0, A_1, \ldots, A_n$ are functions of the (r-1) variables $(x_1,x_2,\ldots,x_{r-1})$.

We assume a lexicographic ordering among the r variables $x_1,\ldots,x_r$ as follows: the variable x_i precedes x_j for i>j. This ordering is useful for realizing operations on MVPs.

The representation of a MVP consists of two lists: a coefficient list PC and a degree list PD. PC consists of the coefficients of the terms in the MVP. Each component in the degree list is the degrees of the individual variables in a term of the MVP in the order specified.

Example

$$P(x_3,x_2,x_1) = 2x_3x_2x_1 - x_3x_2 - x_3x_1 + 2x_3 - x_3x_2x_1 + x_3x_2 + x_3x_1 + x_3 + 3x_2x_1 + 2x_2 + 2x_1 + 1$$

The PC list and PD list representation for this MVP can be written as:

PC $\leftarrow$ (2,-1,-1,2,-1,1,1,1,3,2,2,1)

PD $\leftarrow$ {(2,1,1), (2,1,0), (2,0,1), (2,0,0), (1,1,1), (1,1,0), (1,0,1), (1,0,0), (0,1,1), (0,1,0), (0,0,1), (0,0,0)}

We consider here the problem of evaluating a MVP at a given set of points $(x_r, x_{r-1}, \ldots, x_1)$. Treating a MVP as a single-variable polynomial in one of the variables with coefficients from the set of the remaining variables (as written in (1) above), MVP evaluation consists of two basic steps:

(1) computing b_i = the value of A_i at $(x_{r-1}, x_{r-2}, \ldots, x_1)$ for $i = 0$ to n where n is the degree of the MVP considered as a polynomial in x_r.

(2) computing the value of the single variable polynomial (SVP) $b_0 + b_1 x_r + b_2 x_r^2 + \ldots + b_n x_r^n$ at $x_r = X_r$.

Step (1) is the evaluation of (n+1) MVPs A_i (i=0 to n) in (r-1) variables which can go in parallel. Thus we get a recursive parallel algorithm. See also Hyafil [24] for an alternative treatment of the parallel evaluation of a MVP.

Essentially, the algorithm consists of:

(a) constructing a multiway tree in which the root node represents the given MVP, its children represent the powers of x_r, children of each (power) node represent the powers of x_{r-1} etc.

(b) Evaluating the SVP at the leaves at $x_1 = X_1$; this creates SVPs at the next higher level which can be evaluated at $x_2 = X_2$ etc. At the last stage the SVP in x_r is evaluated at $x_r = X_r$.

The example MVP given above can be written as,

$$P(X_3,X_2,X_1)=X_3(X_2(2X_1-1)+1(-X_1+2))+X_3(X_2(-X_1+1)+(X_1+1))$$
$$+1(X_2(3X_1+2)+(2X_1+1))$$

Its tree structure would be as shown in Figure 7.1.

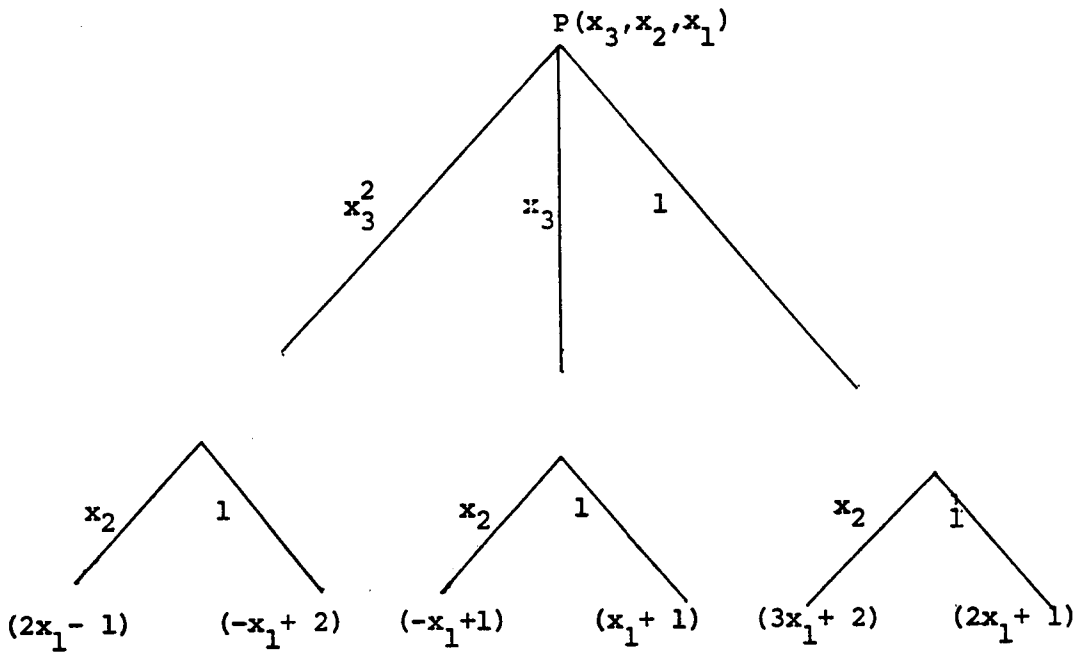

Figure 7.1 MVP Tree.

The parallelism in the algorithm is derived from this tree structure.

We now present the parallel algorithm. It is the same as the sequential algorithm except that the different MVP evaluations can go in parallel. Hence there is no need to give the sequential algorithm as such.

```
FUNCTION MVPVAL(r,pos)
 (* r is the no. of variables in the MVP in this invocation of
 MVPVAL; pos gives the position of the elements in the
(PC,PD) list *)
```

```
IF r=0 THEN (* the no. of variables is 0. So the
              coefficients are returned as values *)
      MVPVAL:=PC(pos)
ELSE (* the no. of variables is≥1.  Evaluates the
      MVPS in (r-1) variables appropriately and
      evaluates the SVP in the r-th  variable *)
       BEGIN
          DEG :=degree of the r-th variable from PD(pos);
          Posn(0):=pos;
          FOR i:=1 TO DEG DO
                 posn(i):=Position from where the term with
                          degree(DEG-i) for the r-th variable
                          is stored in PC;
          FOR i:=0 TO DEG DO IN PARALLEL
                          B(i):=MVPVAL(r-1,Posn(i));
          MVPVAL:=SVPVAL(B,DEG,X_r)
       END
END MVPVAL;
```

Note

Here SVPVAL(B,DEG,X_r) is assumed to be a function procedure for evaluating at X_r a SVP whose coefficients are stored as a vector B, and whose degree is DEG.

The initial invocation of MVPVAL for a polynomial with r variables is MVPVAL(r,1).

Discussion

As noted earlier, the parallelism in this algorithm is due to the inherent tree structure in the representation of a MVP. This tree structure is a generalization of the tree structure of a single variable polynomial.

Note

A SVP of the form, $P(x)=a_0+a_1x+a_2x^2+\ldots+a_nx^n$ can be represented as shown in Figure 7.2.

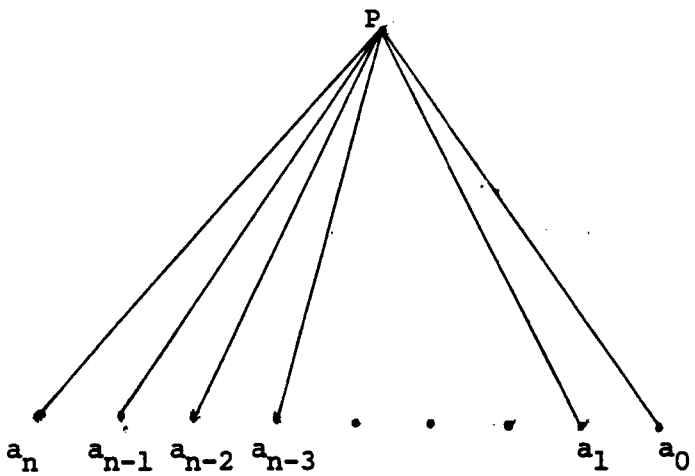

Figure 7.2 SVP Tree.

The evaluation at a given point x=X is realizable in parallel by grouping pairs of branches and evaluating them in parallel and combining later. The evaluation of SVPs occuring in the MVPEVAL algorithm can use this parallelism.

The number of levels in the tree is the number of variables in the MVP and the number of nodes is the number of SVP evaluations to be done to evaluate the MVP. The flow

of computation (corresponding to the two basic steps of algorithm) is from the root node to the leaves for constructing the tree and from the leaves to the root node when the evaluations begin. Thus this is a synchronous tree algorithm. Correctness of this algorithm follows from the observations below:

(1) As a sequential algorithm it implements the two basic steps for MVP evaluation.

(2) In the parallel version no independent task interferes with another independent task.

Complexity

Let there be r variables $(x_r, x_{r-1}, \ldots, x_1)$ in P with the ordering $x_r > x_{r-1} > \ldots > x_1$.

Let the degree of P considered as a polynomial in x_r be d_r. Then there are (d_r+1) MVPs in (r-1) variables associated with the powers of x_r.

Considered as polynomials in x_{r-1}, let the degrees of these polynomials be $d_{r-1}(0), d_{r-1}(1), \ldots, d_{r-1}(d_r)$.

Associated with each of these are $[d_{r-1}(i_1)+1]$ polynomials in (r-2) variables. So, there are $\sum_{i_1=0}^{d_r} [d_{r-1}(i_1)+1]$ polynomials in (r-2) variables which have to be evaluated. Let the degrees of these polynomials be

$$d_{r-2}(i_1, i_2) \quad i_1 = 0 \text{ to } d_r \text{ and } i_2 = 0 \text{ to } d_{r-1}(i_1).$$

Then, there are

$$\sum_{i_1=0}^{d_r} \sum_{i_2=0}^{d_{r-1}(i_1)} [d_{r-2}(i_1, i_2)+1]$$

polynomials in (r-3) variables, each of degree $d_{r-3}(i_1, i_2, i_3)$ for $i_3 = 0$ to $d_{r-2}(i_1, i_2)$.

With this notation, in general, there are

$$\sum_{i_1=0}^{d_r} \sum_{i_2=0}^{d_{r-1}^{(i_1)}} \cdots \sum_{i_{r-j-1}=0}^{d_{j+2}^{(i_1,\ldots,i_{r-j-2})}} [d_{j+1}(i_1,i_2,i_3,\ldots,i_{r-j-1})+1]$$

polynomials in j variables each of degree $d_j(i_1,i_2,\ldots,i_{r-j})$ for $i_{r-j} = 0$ to $d_{j+1}(i_1,i_2,\ldots,i_{r-j-1})$.

Thus the number of input terms for this algorithm could be exponential, in theory. Since MVP evaluation has to consider every term individually at some time, the algorithm is inherently exponential. With parallelism, an exponential number of processors are required.

But in practice, the number of terms is not so large. The total number of SVP evaluations done is given by the number of nodes in the tree representing the MVP.

In the example that we have considered,

$r=3$, $d_3=2$, $d_2(0)=d_2(1)=d_2(2)=1$,

$d_1(0,0)=d_1(0,1)=d_1(1,0)=d_1(1,1)=d_1(2,0)=d_1(2,1)=1$

The number of 3-variable polynomials is 1, the number of 2-variable polynomials is 3 and the no. of single variable polynomials is 6. The total number of single variable polynomial evaluations is, 6 of degree 1, 3 of degree 1, and 1 of degree 2.

The number of nodes in the tree is the number of SVP evaluations which is 10 and the number of levels is the number of variables which is 3.

CHAPTER 8

CONCLUDING REMARKS

We have studied here some aspects of tree structured parallel algorithms. We have shown how programming methods like divide and conquer naturally generate tree structured algorithms, most of which can be realized in parallel. Thus these partitioning techniques also serve as parallelizing techniques.

In addition to those considered here there are several algorithms reported in the literature which have an underlying tree structure. Even [17] gives two parallel methods of performing external sorting, one of which is a tree structured algorithm. This method is a natural parallelization of the merging method for external sorting which uses 4 tapes and which has a tree structure. Given p processors and 4p tapes, the N records are distributed to 2p tapes in such a way that each of them has approximately N/2p records. Each processor is assigned 4 tapes, 2 with N/2p records and 2 empty. Each processor does the usual tape sorting algorithm. Now, there are p tapes containing N/p records each in sorted order. These are then merged in log p steps.

Benteley and Shamos [8] discuss algorithms for problems dealing with the proximity of N points in Euclidean K-space. The basic strategy is to divide the original problem dealing with N points in K dimensions into two subproblems with lesser number of points in K dimensions and a subproblem in K-1 dimensions. This gives rise to doubly recursive algorithms with a tree structure.

Parallel algorithms for the calculation of convolution of two finite sequences which can be implemented on a tree machine have been described by Jenq [26]. The tree structure of the algorithms is derived from the application of the divide and conquer technique to the problem.

Algorithms for solving recurrences, see, for instance, Kuck [33], evaluation of decision trees as discussed by Kuhn [34], and compiling expressions [33] are some of the other algorithms with a tree structure.

Horowitz and Sahni [22] have observed that recursive programs are often easier to prove correct than the corresponding iterative versions. This applies to tree structured algorithms as they are essentially recursive algorithms. There have been attempts recently to provide a formal basis to the correctness of parallel programs; see, for instance, Owicki and Gries [42]. Their work extends Hoares's axiom system defined for sequential programs into the realm of parallel programs. Essentially, the approach consists of two steps: (i) Correctness proof of the independent actions considered as sequential programs;

(ii) proof of properties concerning interference (or interaction) between the actions. With tree structured algorithms this task may be somewhat simplified because, usually, the study of interaction between a node and its children provide the pattern for the interaction between any node and its children in the tree.

Another aspect which has received much attention is the architecture suitable for realizing parallel computations. In fact, a classification of parallel algorithms is based on the type of architectures to which they correspond naturally as we saw in Chapter 2. Recently, there have been proposals made by Snyder [46] on reconfigurable parallel architectures based on the systolic machines proposed by Kung [37]. Of particular relevance to the tree structured algorithms discussed here are the tree machines discussed by Benteley and Kung [7], Despain and Patterson [14], Harris and Smith [20], and Horowitz and Zorat [23].

Many languages and language extensions have been proposed to provide support for expressing parallel algorithms. See, for instance, Anderson [4], Conway [12], Lorin [40]. It would be interesting to order the proposed structures for parallel processing with respect to their power and expression as has been done for control structures for sequential processing by Kosaraju [31] and Ledgard [38]. At least for many tree structured algorithms such as the ones studied here the two connectives AND and OR were sufficient and powerful enough to express parallelism

between actions. An issue to be addressed in this regard is at what levels should parallelism be explicitly indicated by the programmer. This is somewhat similar to the question of whether recursion should be explicitly indicated in programs. It may be possible for a compiler to detect parallelism at the expression level or even at statement level but it appears that it may be difficult to identify parallelism at higher levels of abstraction, such as procedures, without explicit identification of parallel actions.

Finally, we note that Bokhari [10] gives an algorithm for the optimum assignment of parallel actions to the different processors in a parallel processor system, for the case of an arbitrary number of processors, provided that the intermodule communication pattern of the program is a tree.

In this thesis we have studied cetain properties of tree-structured parallel algorithms without relating them to specific parallel architectures. Such studies would prove useful in providing a taxonomy of parallel algorithms based on their inherent structure. They would also be useful in identifying programming paradigms for developing parallel algorithms. Concerning the performance of a tree-structured parallel algorithm we have shown that the speed-up in going from sequential to parallel execution is given by the ratio of the number of nodes in the tree to the number of levels in the tree. For an algorithm whose structure is a complete binary tree the speed-up, in general, is by a factor of

$O(n/\log_2 n)$ as is the case with three of the four case-study algorithms discussed here. This indicates the inherent limitation in speed-up for many parallel algorithms with a tree structure.

Parallel computation is one of the areas of computer science that is growing in practical importance. The design of parallel algorithms has far more complexities than the design of sequential algrithms, which is itself complicated enough to remain as a highly creative activity. A methodology for developing parallel algorithms is yet to emerge. We believe that studying the inherent structure of parallel algorithms would be a contribution to this area of knowledge.

LIST OF REFERENCES

LIST OF REFERENCES

1. Ackermann,W.B., Data Flow Languages., Proc.1979 National Comput.Conf., 48, June 1979, pp.1087-1095.

2. Aho,A.V., Hopcroft,J.E., Ullman,J.D., The Design And Analysis of Computer Algorithms., Addison-Wesley, 1975.

3. Anderson,G.A., Jensen,E.D., Computer Interconnection Structures: Taxonomy, Characteristics, and Examples., ACM Comput.Surv., 7(4), 1975, pp.197-213.

4. Anderson,J.P., Program Structures for Parallel Processing., CACM, 8(12), pp.786-788.

5. Bayer,R., McCreight,E., Organisation and Maintenance of Large Ordered Indexes., Acta Inf., 1(3), 1972, pp.173-189.

6. Bayer,R., Schkolnick,M., Concurrency of Operations on B-Trees., Acta inf., 9(1), 1977, pp.1-21.

7. Bentley,J.L., Kung,H.T., A Tree Machine for Searching Problems., Proc. 1979 Int.Conf. on Parallel Processing, IEEE, New York, pp.257-266.

8. Bentley,J.L., Shamos,M.I., Divide and Conquer in Multidimensional Space., Proc. 8th Annual ACM Symp. on Theory of Computing, 1976, pp.220-230.

9. Bernstein,P.A., Goodman,N., Concurrency Control in Distributed Database Systems., ACM Comput.Surv., 13(2), June 1981, pp.185-221.

10. Bokhari,S.H., A Shortest Tree Algorithm for Optimal Assignments across Space and Time in a Distributed Processor Systems., IEEE Trans.Software Engg., Vol.SE7(6), November 1981, pp.583-589.

11. Borodin,A., Munro,I., The Computational Complexity of Algebraic and Numeric Problems., American Elsevier, New York,1975.

12. Conway,M.E., A Multiprocessor System Design., Proc.FJCC, 24, 1963.

13. Dennis,J.B., Misunas,D.P., A Preliminary Architecture for a Basic Data Flow Processor., Proc.IEEE 2nd Annual Symp. Comp. Architecture, 1975, pp.126-132.

14. Despain,A., Patterson,D., X-Tree: A Structured Multiprocessor Computer Architecture., Proc.IEEE 5th Annual Symp. Comp. Architecture, April 1978, pp.144-151.

15. Dijkstra,E.W., Lamport,L., Martin,A.J., Shcolten,C.S., Steffen,E.F.M., On the Fly Garbage Collection: An Exercise in Cooperation., CACM, 21(11), 1978, pp.966-976.

16. Ellis,C.S., Concurrent Search And Insertion in AVL Trees., Proc.1979 Int.Conf. on Parallel Processing, IEEE, pp.250-256.

17. Even,S., Parallelism in Tape Sorting., CACM, April 1974, pp.202-204.

18. Flynn,M.J., Very High Speed Computing Systems., Proc. IEEE, Vol.54, December 1966, pp.1901-1909.

19. Gilmore,P.A., Structuring of Parallel Algorithms., JACM, Vol.15, 1968, pp.176-192.

20. Harris,J., Smith,D., Simulation Experiments of a Tree Organized Multicomputer., Proc.IEEE 6th Annual Symp. Comp. Architecture, April 1979, pp.83-89.

21. Hirschberg,D.S., A Linear Space Algorithm for Computing Maximal Common Subsequences., CACM, 18(6), June 1975, pp.341-343.

22. Horowitz,E., Sahni,S., Fundamentals of Computer Algorithms., Computer Science Press, 1978.

23. Horowitz,E., Zorat,A., The Binary Tree as an Interconnection Network: Applications to Multiprocessor Systems and VLSI., IEEE Trans.Comput., C30(4), April 1981, pp.247-253.

24. Hyafil,L., On the Parallel Evaluation of Multivariate Polynomials., SIAM J.Comput., 8(2), May 1979, pp.120-123.

25. Hyafil,L., Kung,H.T., The Complexity of Parallel Evaluation of Linear Recurrences., JACM, 24(3), July 1977, pp.513-521.

26. Jenq,Y.-C., Digital Convolution Algorithm for Pipelining Multiprocessor Systems., IEEE Trans.Comput., C30(12), December 1981, pp.966-973.

27. Knuth,D.E., The Art of Computer Programming., Vol.1, Fundamental Algorithms, Addison-Wesley, 1969.

28. Knuth,D.E., The Art of Computer Programming., Vol.3, Sorting and Searching, Addison-Wesley, 1973.

29. Knuth,D.E., Big Omicron, Big Omega and Big Theta., SIGACT News, ACM, April 1976.

30. Kogge,P.M., Parallel Solution of Recurrence Problems., IBM J. of Research and Development, Vol.18, 1974, pp.138-148.

31. Kosaraju,R., Analysis of Structured Programs., J. Comput. and Syst. Sci., 9(3), December 1974, pp.232-255.

32. Krishnamurthy,E.V., Venkateswaran,H., A Parallel Wilf Algorithm for Complex Zeros of a Polynomial., BIT 21, 1981, pp.104-111.

33. Kuck,D.J., Parallel Processing of Ordinary Programs., Adv,, in Computers, Vol.15, 1976, pp.119-176.

34. Kuhn,R.H., Fast Evaluation of Arbitrary Decision Trees., Int. Conf. on Parallel Processing, 1979, pp.267-268.

35. Kung,H.T., New Algorithms and Lower Bounds for the Parallel Evaluation of Certain Rational Expressions and Recurrences., JACM, 23(2), April 1976, pp.252-261.

36. Kung,H.T., Some Complexity Bounds for Parallel Computation., Proc. 6th Annual ACM Symp. on Theory of Computing, May 1974, pp.323-333.

37. Kung,H.T., The Structure of Parallel Algorithms., Adv. in Computers, Vol.19, 1980, pp.65-112.

38. Ledgard,H.F., Marcotty,M., A Genealogy of Control Structures., CACM, 18(11), November 1975, pp.629-639.

39. Lint,B., Agerwala,T., Communication Issues in the Design and Analysis of Parallel Algorithms., IEEE Trans.Software Engg., SE7(2), March 1981.

40. Lorin,H., Parallelism in Hardware and Software: Real and Apparent Concurrency., Pr. Hall Series in A.C., 1972.

41. Nievergelt,J., Reingold,E.M., Binary Search Trees of Bounded Balance., Proc. 4th Annual ACM Symp. on Theory of Computing, 1972, pp.137-142.

42. Owicki,S., Gries,D., Verifying Properties of Parallel Programs., CACM, May 1976, 19(5), pp.279-285.

43. Samadi,B., B-Trees in a System with Multiple Users., Inf. process. Lett., 5(4), 1976, pp.107-112.

44. Savage,J.E., The Complexity of Computing., Wiley-Interscience, 1976.

45. Savage,J.E., Swamy,S., Space-Time Trade-offs on the FFT Algorithms., IEEE Trans.Inf.Theory, IT24(5), September 1978, pp.563-568.

46. Snyder,L., Introduction to the Configurable, Highly Parallel Computer., Purdue University Report, CSD-TR-351, May 1981.

47. Stone,H.S., Introduction to Computer Organization and Data Structures., McGraw-Hill, 1972.

48. Stone,H.S., Problems of Parallel Computation, in Complexity of Sequential and Parallel Numerical Algorithms', ed. by J.F. Traub, Academic Press, New York, 1973, pp.1-16.

49. Valiant,L.G., Parallelism in Comparison Problems., SIAM J. Comput., 4(3), September 1975, pp.348-355.

50. Wagner,R.A., Fischer,M.J., The String-to-String Correction Problems., JACM, 21(1), January 1974, pp.168-173.

51. Weide,B., A Survey of Analysis Techniques for Discrete Algorithms., ACM Comput.Surv., 9(4), December 1977, pp.291-313.

52. Wilf,H.S., A Global Bisection Algorithm for Computing the Zeros of Polynomials in the Complex Plane., JACM, 25(3), July 1978, pp.415-420.

53. Wittie,L.D., Communication Structure for Large Networks of Microcomputers., IEEE Trans.Comput., C30(4), April 1981, pp.264-273.

APPENDIX

APPENDIX

A BIBLIOGRAPHY FOR PARALLEL ALGORITHMS

A.1 Graph Algorithms

Dekel,E., Nassimi,D., Sahni,S., Parallel Matrix and Graph Algorithms., SIAM J.Comput., 10(4), November 1981, pp.657-675.

Deo,N., Pang,C.Y., Lord,R.E., Two Parallel Algorithms for Shortest Path Problems., Proc. 1980 Int.Conf. on Parallel Processing, IEEE, pp.244-253.

Goodman,S.E., Hedetniemi,S.T., Introduction to the Design and Analysis of Algorithms., McGraw-Hill, 1977.

Hirschberg,D.S., Parallel Algorithms for the Transitive Closure and the Connected Components., Proc. 8th Annual ACM Symp. on Theory of Computing, 1976, pp.55-57.

Hirschberg,D.S., Chandra,A.K., Sarwate,D.V., Computing Connected Components on Parallel Computers., CACM, 22(8), August 1979, pp.461-464.

Ja 'Ja',J., Simon,J., Planarity Testing., Math. found. of CS1980, Proc. of 9th Symp., Lecture notes in C.S.88, pp.305-319.

Nassimi,D., Sahni,S., Finding Connected Components and Connected ones on a Mesh-connected Parallel Computer., SIAM J. Comput., 9(4), November 1980, pp.744-757.

Reghbati,E., Corneil,D.G., Parallel Computations in Graph Theory., SIAM J. Comput., 7(2), May 1978, pp.230-237.

Savage,C., Ja'Ja',J., Fast, Efficient Parallel Algorithms for Some Graph Problems., SIAM J.Comput., 10(4), November 1981, pp.682-691.

A.2 Sorting Algorithms

Baudet,G.M., Stevenson,D., Optimal Sorting Algorithms for Parallel Computers., IEEE Trans.Comput., C27(1), January 1978, pp.84-87.

Gavril,F., Merging with Parallel Processors., CACM, 18(10), October 1975, pp.588-591.

Hirschberg,D.S., Fast Parallel Sorting Algorithms., CACM, 21(8), August 1978, pp.657-661.

Nassimi,D., Sahni,S., Bitonic Sort on a Mesh-connected Parallel Computer., IEEE Trans.Comput., Vol.28, 1979, pp.2-7.

Preparata,F.P., Parallelism in Sorting., Proc. 1977 Int. Conf. on Parallel Processing, IEEE, pp.202-206.

Thompson,C.D., Kung,H.T., Sorting on a Mesh-connected Parallel Computer., CACM, 20(4), 1977, pp.263-271.

Valiant,L.G., Parallelism in Comparison Problems., SIAM J. Comput., 4(3), September 1975, pp.348-355.

A.3 Compiling Algorithms

Barak,A., Shamir,E., On the Parallel Evaluation of Boolean Expressions., SIAM J.Comput., 5(4), December 1976, pp.678-681.

Brent,R.P., The Parallel Evaluation of General Arithmetic Expressions., JACM, 21(2), 1974, pp.201-206.

Hellerman,H., Parallel Processing of Algebraic Expressions., IEEE Trans.Comput., EC15, 1966, pp.82-91.

Kuck,D.J., Maruyama,K., Time Bounds on the Parallel Evaluation of Arithmetic Operations., SIAM J. Comput., 4(2), June 1975, pp.147-162.

Ramamoorthy,C.V., Gonzalez,M.J., A Survey of Recognising Parallel Processable Streams in Computer Programs., AFIPS Conf. Proc., FJCC, 1969, Vol.35, pp.1-15.

A.4 Numerical Algorithms

Baudet,G.M., Asynchronous Iterative Methods for Multiprocessors., JACM, 25(2), 1978a, pp.226-244.

Chen,S.C., Time and Parallel Processor Bounds for Linear Recurrence Systems with Constant Coefficients., Proc. 1976 Int.Conf. on Parallel Processing, IEEE, pp.196-205.

Csanky,L., Fast Parallel Matrix Inversion Algorithms., SIAM J. Comput., 5(4), December 1976, pp.618-623.

Gajski,D.D., An Algorithm for Solving Linear Recurrence Systems on Parallel and Pipelined Machines., IEEE Trans. Comput., C30(3), March 1981, pp.190-206.

Gentleman,W.M., Some Complexity Results for Matrix Computations on Parallel Processors., JACM, 25(1), 1973, pp.112-116.

Heller,D., A Determinant Theorem with Applications to Parallel Algorithms., SIAM J. of Numerical Analysis, Vol.11, 1974, pp.559-568.

Heller,D., A Survey of Parallel Algorithms in Numerical Linear Algebra., SIAM Rev., 20(4), October 1978, pp.740-777.

Kogge,P.M., Parallel Solution of Recurrence Problems., IBM J. of Research and Development, Vol.18, 1974, pp.138-148.

Krishnamurthy,E.V., Venkateswaran,H., A Parallel Wilf Algorithm for Complex Zeros of a Polynomial., BIT 21, 1981, pp.104-111.

Kuck,D.J., Sameh,A., Parallel Computation of Eigenvalues of Real Matrices., INFO71, pp.1266-1272.

Maruyama,K., On the Parallel Evaluation of Polynomials., IEEE Trans.Comput., C22, January 1973, pp.2-5.

Miranker, W.L., A Survey of Parallelism in Numerical Analysis., SIAM Rev., Vol.13, 1971, pp.524-547.

Munro,I., Paterson,M., Optimal Algorithms for Parallel Polynomial Evaluation., J. Comput. System Sci., Vol.7, 1973, pp.189-198.

Pease., An Adaptation of the FFT for Parallel Processing., JACM, Vol.15, April 1968, pp.252-264.

Stone,H.S., An Efficient Parallel Algorithm for the Solution of Tridiagonal Linear Equations., JACM, 1973.

A.5 Other Algorithms

Arcelli,C., Levialdi,S., Parallel Shrinking in Three Dimensions., Computer Graphics and Image Processing, Vol.4, 1972, pp.21-30.

Carrol,A.B., Wetherald,R.T., Application of Parallel Processing to Numerical Weather Prediction., JACM, Vol.14, 1967, pp.591-614.

Kung,H.T., Song,S,W., A Parallel Garbage Collection Algorithm and its Correctness Proof., Proc. 18th Annu. Symp. Found. Comput. Sci., 1977, pp.120-131.

Ladner,R., Fischer,M., Parallel Prefix Computation., Int. Conf. on Parallel Processing, 1977, pp.218-223.

Lamport,L., Garbage Collection with Multiple Processes: An Exercise in Parallelism., Proc. IEEE Conf. on Parallel Processi ng, 1976, pp.50-54.

Steele,G.L., Multiprocessing Compactifying Garbage Collection., CACM, 18(9), 1975, pp.125-143.

Stone,H.S., Parallel Processing with the Perfect Shuffle., IEEE Trans.Comput., C20(2), February 1971, pp.153-161.

VITA

H. Venkateswaran was born in Kerala State,India on November 13, 1951. He graduated from St. Joseph's High School, Bangalore, India in 1966. In 1970, he graduated from St. Joseph's College, Bangalore earning his Bachelor of Science degree.

He worked as a programmer in the National Aeronautical Laboratory, Bangalore, India for ten years from August 1970 to August 1980. He earned his Master's degree in Mathematics from Bangalore University in 1976.

He entered the Graduate School of The University of Tennessee at Knoxville in September 1980 as a Graduate Teaching Assistant in the Department of Computer Science. After graduating from The University of Tennessee, he plans to pursue his Ph.D. in Computer Science.